

University of Warwick institutional repository: <http://go.warwick.ac.uk/wrap>

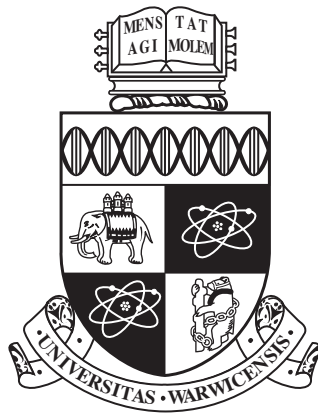
A Thesis Submitted for the Degree of PhD at the University of Warwick

<http://go.warwick.ac.uk/wrap/66022>

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.



Monitoring, Analysis and Optimisation of I/O in Parallel Applications

by

Steven Alexander Wright

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

July 2014

Abstract

High performance computing (HPC) is changing the way science is performed in the 21st Century; experiments that once took enormous amounts of time, were dangerous and often produced inaccurate results can now be performed and refined in a fraction of the time in a simulation environment. Current generation supercomputers are running in excess of 10^{16} floating point operations per second, and the push towards exascale will see this increase by two orders of magnitude. To achieve this level of performance it is thought that applications may have to scale to potentially billions of simultaneous threads, pushing hardware to its limits and severely impacting failure rates.

To reduce the cost of these failures, many applications use checkpointing to periodically save their state to persistent storage, such that, in the event of a failure, computation can be restarted without significant data loss. As computational power has grown by approximately $2\times$ every 18 – 24 months, persistent storage has lagged behind; checkpointing is fast becoming a bottleneck to performance.

Several software and hardware solutions have been presented to solve the current I/O problem being experienced in the HPC community and this thesis examines some of these. Specifically, this thesis presents a tool designed for analysing and optimising the I/O behaviour of scientific applications, as well as a tool designed to allow the rapid analysis of one software solution to the problem of parallel I/O, namely the parallel log-structured file system (PLFS). This thesis ends with an analysis of a modern Lustre file system under contention from multiple applications and multiple compute nodes running the same problem through PLFS. The results and analysis presented outline a framework through which application settings and procurement decisions can be made.

This thesis is dedicated to the memory of my Grandad.

Ian Haig Henderson

(1928 – 2014)

Acknowledgements

Since walking into the Department of Computer Science for the first time in October 2006, I have been fortunate enough to meet, work with and enjoy the company of many special people. First and foremost, I would like to thank my supervisor, Professor Stephen Jarvis, for all his help and hard work over the past 4 years and for allowing me the opportunity to undertake a Ph. D. I would also like to thank him for providing me with funding and a post-doctoral position in the department.

Secondly, I would like to thank the two best office-mates I'm ever likely to have, Dr. Simon Hammond and Dr. John Pennycook. I have been in a lull since each of them moved on to pastures new. I consider my time sharing an office with each of them to be both the most productive (with Si) and most entertaining (with John) time in my Warwick career, and I miss their daily company dearly.

Thirdly, I acknowledge my current office-mates, Robert Bird and Richard Bunt. Both provide nice light entertainment and interesting discussions to break up the day and make my working environment a more pleasant place to spend my time. Additionally, I thank the other members of the High Performance and Scientific Computing group, past and present – David Beckingsale, Dr. Adam Chester, Peter Coetzee, James Davis, Timothy Law, Andy Mallinson, Dr. Gihan Mudalige, Dr. Oliver Perks and Stephen Roberts – for their lunchtime company and occasional pointless discussions.

Finally, within the University, I would like to thank the group of individuals that have helped me through both my undergraduate and postgraduate studies – Dr. Abhir Bhalerao, Jane Clarke, Dr. Matt Ismail, Dr. Arshad Jhumka, Dr. Matthew Leeke, Dr. Christine Leigh, Prof. Chang-Tsun Li, Rod Moore, Dr. Roger Packwood, Catherine Pillet, Jackie Pinks, Gill Reeves-Brown, Phillip

Taylor, Stuart Valentine, Dr. Justin Ward, Paul Williamson, amongst many others.

Outside of University, thanks go to the organisations that have contributed resources and expertise to much of the material in this thesis: the team at the Lawrence Livermore National Laboratory for allowing access to the Sierra and Cab supercomputers; the team at Daresbury Laboratory for granting time on their IBM BlueGene/P; and finally, to both Meghan Wingate-McLelland, at Xyratex, and John Bent, at EMC², for contributing their time and expertise to my investigations into the parallel log-structured file system.

Special thanks is reserved for my closest friend for four years of undergraduate studies and current snowboarding partner, Chris Straffon. I only wish our snowboarding excursions were both longer and more frequent; they're currently the week of the year I look forward to the most.

And last, but certainly not least, thanks go to my family – Mum, Dad, Gemma and Paula; my beloved Gran and Grandad; my aunties and uncles; and my nieces and nephews – Chloe, Sophie, Megan, Lauren, Charlie, Holly-Mae, Liam, Tyler, Aimee and Isabelle – who often make me laugh uncontrollably and cost me so much money each Christmas time. And finally huge thanks go to my girlfriend, Jessie, for putting up with me throughout my Ph. D. and making my life more enjoyable.

Declarations

This thesis is submitted to the University of Warwick in support of the author's application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree.

The work presented (including data generated and data analysis) was carried out by the author except in the cases outlined below:

- Performance data in Chapters 4 and 5 for the Sierra supercomputer were collected by Dr. Simon Hammond.

Parts of this thesis have been previously published by the author in the following publications:

- [141] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. H. Bhalerao, and S. A. Jarvis. Parallel File System Analysis Through Application I/O Tracing. *The Computer Journal*, 56(2):141–155, February 2013
- [142] S. A. Wright, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. Lightweight Parallel I/O Analysis at Scale. *Lecture Notes in Computer Science (LNCS)*, 6977:235–249, October 2011
- [143] S. A. Wright, S. D. Hammond, S. J. Pennycook, I. Miller, J. A. Herdman, and S. A. Jarvis. LDPLFS: Improving I/O Performance without Application Modification. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium Workshops & PhD Forum (IPDPSW'12)*, pages 1352–1359, Shanghai, China, 2012. IEEE Computer Society, Washington, DC
- [144] S. A. Wright, S. J. Pennycook, S. D. Hammond, and S. A. Jarvis. RIOT – A Parallel Input/Output Tracer. In *Proceedings of the 27th Annual UK*

Performance Engineering Workshop (UKPEW'11), pages 25–39, Bradford, UK, July 2011. The University of Bradford, Bradford, UK

- [145] S. A. Wright, S. J. Pennycook, and S. A. Jarvis. Towards the Automated Generation of Hard Disk Models Through Physical Geometry Discovery. In *Proceedings of the 3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS'12)*, pages 1–8, Salt Lake City, UT, November 2012. ACM, New York, NY

In addition, research conducted during the period of registration has also led to the following publications which, while not directly focused on parallel I/O, have nevertheless shaped the research presented in this thesis:

- [12] R. F. Bird, S. J. Pennycook, S. A. Wright, and S. A. Jarvis. Towards a Portable and Future-proof Particle-in-Cell Plasma Physics Code. In *Proceedings of the 1st International Workshop on OpenCL (IWOCL'13)*, Atlanta, GA, May 2013. Georgia Institute of Technology, GA
- [13] R. F. Bird, S. A. Wright, D. A. Beckingsale, and S. A. Jarvis. Performance Modelling of Magnetohydrodynamics Codes. *Lecture Notes in Computer Science (LNCS)*, 7587:197–209, July 2013
- [98] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Wright, and S. A. Jarvis. On the Acceleration of Wavefront Applications using Distributed Many-Core Architectures. *The Computer Journal*, 55(2):138–153, February 2012
- [99] S. J. Pennycook, S. D. Hammond, S. A. Wright, J. A. Herdman, I. Miller, and S. A. Jarvis. An Investigation of the Performance Portability of OpenCL. *Journal of Parallel and Distributed Computing (JPDC)*, 73(11):1439–1450, November 2013

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- The University of Warwick, United Kingdom:
Engineering and Physical Sciences Research Council Doctoral Training
Accounts Studentship (2010–2014)
- Royal Society:
Industry Fellowship Scheme (IF090020/AM)
- UK Atomic Weapons Establishment:
“The Production of Predictive Models for Future Computing
Requirements” (CDK0660)
“AWE Technical Outreach Programme” (CDK0724)
“TSB Knowledge Transfer Partnership” (KTP006740)

Abbreviations

ADIO	Abstract Device Interface for Parallel I/O
ANL	Argonne National Laboratory
API	Application Programmable Interface
ATA	Advanced Technology Attachment
B/W	Bandwidth
BG/P	IBM BlueGene/P
CPU	Central Processing Unit
DFS	Distributed File System
EMC²	EMC Corporation
FLOP/s	Floating-Point Operations per Second
FUSE	File system in User Space
GB	1024 ³ Bytes
GFLOP/s	10 ⁹ FLOP/s
GPFS	IBM General Parallel File System
HDD	Hard Disk Drive
HDF-5	Hierarchical Data Format, Version 5
HPC	High-Performance Computing
I/O	Input/Output
KB	1024 Bytes
LANL	Los Alamos National Laboratory
LBNL	Lawrence Berkeley National Laboratory
LDPLFS	1d Loadable PLFS
LLNL	Lawrence Livermore National Laboratory
MB	1024 ² Bytes
MDS	Metadata Server
MDT	Metadata Target

MFLOP/s	10^6 FLOP/s
MGS	Management Server
MPI	Message Passing Interface
NASA	National Aeronautics and Space Administration
NFS	Network File System
NPB	NASA Parallel Benchmarks
OCF	Open Compute Facility
OSS	Object Storage Server
OST	Object Storage Target
PFLOP/s	10^{15} FLOP/s
PLFS	Parallel Log-Structured File System
PMPI	Profiling Message Passing Interface
POSIX	Portable Operating System Interface
PVFS	Parallel Virtual File System
QDR	Quad Data Rate
RAID	Redundant Array of Inexpensive/Independent Disks
RIOT	RIOT I/O Toolkit
RPM	Revolutions per Minute
SAS	Serial Attached SCSI
SATA	Serial-ATA
SSD	Solid State Drive
STFC	Science & Technology Facilities Council
TB	1024^4 Bytes
TFLOP/s	10^{12} FLOP/S
UFS	UNIX File System
ZBR	Zoned-Bit Recording

Contents

Abstract	ii
Dedication	iii
Acknowledgements	iv
Declarations	vi
Sponsorship and Grants	viii
Abbreviations	ix
List of Figures	xvii
List of Tables	xxi
1 Introduction	1
1.1 Motivation	2
1.2 Thesis Contributions	2
1.3 Thesis Overview	4
2 Performance Analysis and Engineering	7
2.1 Parallel Computation	8
2.2 I/O in Parallel Computing	10
2.2.1 Issues in Parallel I/O	10
2.2.2 Parallel File Systems	12
2.2.3 Parallel I/O Middleware	15
2.3 Performance Engineering Methodologies	17
2.3.1 Benchmarking	18
2.3.2 System Monitoring and Profiling	19

2.3.3	Analytical Modelling	21
2.3.4	Simulation-based Modelling	22
2.4	Summary	24
3	Hardware and Software Overview	26
3.1	Hard Disk Drive	26
3.1.1	Disk Drive Mechanics	26
3.1.2	Data Layout	27
3.1.3	Disk Controller	30
3.1.4	Redundant Array of Independent Disks	31
3.2	File Systems	33
3.2.1	The Extended File System	33
3.2.2	The Sun Network File System	36
3.3	Distributed File Systems	37
3.3.1	The Lustre File System	39
3.3.2	IBM's General Parallel File System	40
3.3.3	The Parallel Log-structured File System	41
3.4	Computing Platforms	42
3.5	Input/Output Benchmarking Applications	45
3.6	Summary	47
4	I/O Tracing and Application Optimisation	49
4.1	The RIOT I/O Toolkit	50
4.1.1	Feasibility Study	53
4.2	File System Analysis	55
4.2.1	Distributed File Systems – Lustre and GPFS	56
4.3	Middleware Analysis and Optimisation	61
4.4	Summary	65
5	Analysis and Rapid Deployment of the Parallel Log-Structured File System	68

5.1	Analysis of PLFS	69
5.2	Rapid Deployment of PLFS	71
5.2.1	Performance Analysis	74
5.3	Summary	81
6	Parallel File System Performance Under Contention	82
6.1	Effective Use of Uncontended Parallel File Systems	83
6.2	Quantifying the Performance of Contended File Systems	85
6.3	Performance Comparison: Lustre vs. PLFS	90
6.4	Summary	93
7	Discussion and Conclusions	96
7.1	Limitations	98
7.2	Future Work	100
7.3	The Road to Exascale	101
	Bibliography	105
	Appendices	127
A	RIOT Feasibility Study – Additional Results	127
B	Numeric Data for Perceived and Effective Bandwidth	130
C	FLASH-IO Analysis and Optimisation Data	133
D	LDPLFS Source Code Examples	135
E	LDPLFS Numeric Data	137
F	Optimality Search Numeric Data	141
G	PLFS Performance and Stripe Collision Data	142

List of Figures

2.1	An example of the parallelisation of a simple particle simulation between four processors.	9
2.2	The three basic approaches to I/O in parallel applications.	11
2.3	An example of two nodes (four ranks per node) writing to a file system with collective buffering off and on.	16
3.1	Basic internal structure of a hard disk drive.	27
3.2	Data layout on a disk with no zoning and three zones of increasing density.	28
3.3	Four examples of serpentine sector mapping.	29
3.4	An example of four requests fulfilled in-order without NCQ and out of order with NCQ.	32
3.5	Two common RAID data distribution schemes.	33
3.6	Structure of an ext2 inode block.	34
3.7	An example Lustre configuration with four OSSs and a fail-over MGS and MDS.	38
3.8	An example of a GPFS setup with four OSSs connected via a high performance switch to three targets and separate management and metadata targets.	40
3.9	An application's view of a file and the underlying PLFS container structure.	41
4.1	Tracing and analysis workflow using the RIOT toolkit.	50

4.2	Total runtime of RIOT overhead analysis benchmark for the functions <code>MPI_File_write_all()</code> and <code>MPI_File_read_all()</code> , on three platforms at varying core counts, with three different configurations: No RIOT tracing, POSIX RIOT tracing and complete RIOT tracing.	55
4.3	User-perceived bandwidth for applications on the three test systems.	57
4.4	Effective POSIX and MPI bandwidth for IOR through MPI-IO. .	59
4.5	Effective POSIX and MPI bandwidth for IOR through HDF-5. .	59
4.6	Effective POSIX and MPI bandwidth for FLASH-IO.	60
4.7	Effective POSIX and MPI bandwidth for BT Problem C, as measured by RIOT.	60
4.8	Percentage of time spent in POSIX functions for FLASH-IO on three platforms.	62
4.9	Composition of a single, collective MPI write operation on MPI ranks 0 and 1 of a two core run of FLASH-IO, called from the HDF-5 middleware library in its default configuration.	63
4.10	Composition of a single, collective MPI write operation on MPI ranks 0 and 1 of a two core run of FLASH-IO, called from the HDF-5 middleware library after data-sieving has been disabled. .	63
4.11	Perceived bandwidth for the FLASH-IO benchmark in its original configuration (Original), with data-sieving disabled (No DS), and with collective buffering enabled <i>and</i> data-sieving disabled (CB and No DS) on Minerva and Sierra, as measured by RIOT. . . .	65
5.1	Concurrent <code>write()</code> operations for BT class C on 256 cores on Minerva and Sierra.	71
5.2	The control flow of LDPLFS in an applications execution. . . .	73
5.3	Benchmarked MPI-IO bandwidths on FUSE, the <code>ad_plfs</code> driver, LDPLFS and the standard <code>ad_ufs</code> driver (without PLFS). . . .	75

5.4	BT benchmarked MPI-IO bandwidths using MPI-IO, as well as PLFS through ROMIO and LDPLFS.	78
5.5	FLASH-IO benchmarked MPI-IO bandwidths using MPI-IO, as well as PLFS through ROMIO and LDPLFS.	80
6.1	Write bandwidth achieved over 1,024 cores by varying just the stripe count and just the stripe size.	84
6.2	Write bandwidth achieved over 1,024 cores by varying both the stripe count and the stripe size.	85
6.3	The performance per-task of the <i>lscratchc</i> file system under contention, with the ideal upper and lower bounds.	88
6.4	Performance of each of 4 tasks over 5 repetitions where all tasks are contending the file system.	89
6.5	Graphical representation of the data in Table 6.3, showing optimal performance at 160 stripes per file, but very minor performance degradation at just 32 stripes per file.	90
6.6	Achieved write bandwidth achieved for IOR through an optimised Lustre configuration and through the PLFS MPI-IO driver. . . .	91
6.7	The number of OST collisions for IOR running through PLFS with 512 cores.	92
7.1	The memory hierarchy with potentially two additional layers for improved I/O performance on supercomputers.	102
A.1	Total runtime of RIOT overhead analysis software for the functions <code>MPI_File_write()</code> and <code>MPI_File_read()</code> , on three platforms, with three different configurations: No RIOT tracing, POSIX RIOT tracing and complete RIOT tracing.	127

A.2	Total runtime of RIOT overhead analysis software for the functions <code>MPI_File_write_at_all()</code> and <code>MPI_File_read_at_all()</code> , on three platforms, with three different configurations: No RIOT tracing, POSIX RIOT tracing and complete RIOT tracing. . . .	128
-----	---	-----

List of Tables

3.1	Hardware specification of the Minerva, Sierra and Cab supercomputers.	42
3.2	Configuration for the GPFS installation connected to Minerva. .	42
3.3	Configuration for the <i>lscratchc</i> Lustre File System installed at LLNL in 2011 (for the experiments in Chapter 5) and 2013 (for the experiments in Chapter 6).	43
3.4	Hardware configuration for the IBM BlueGene/P system at the Daresbury Laboratory.	45
3.5	Configuration for the GPFS installation connected Daresbury Laboratory’s BlueGene/P, where data is first written to Fibre Channel connected disks because being staged to slower SATA disks.	45
5.1	Perceived and Effective Bandwidth (MB/s) for BT class C through MPI-IO and PLFS, as well as the speed-up generated by PLFS. .	70
5.2	Time in seconds for UNIX commands to complete using PLFS through LDPLFS, and without PLFS.	76
6.1	IOR configuration options for experiments.	83
6.2	The average number of OSTs in use and their average load based on the number of concurrent I/O intensive jobs.	87
6.3	Average and total bandwidth achieved across four tasks for a varying stripe size request, along with values for the average number of tasks competing for 1, 2, 3 and 4 OSTs respectively. . . .	89
6.4	OST usage and average load for the Stampede I/O setup described by Behzad et al. [10].	90

6.5	Stripe collision statistics for PLFS backend directory running with 4,096 cores.	93
A.1	Incidence of <code>MPI_File_*</code> function calls in 9 application suites, benchmarks and I/O libraries.	128
A.2	Average time (s) to perform one hundred 4 MB operations: with- out RIOT, with only POSIX tracing and with complete MPI and POSIX RIOT tracing. The change in time is shown between full RIOT tracing and no RIOT tracing.	129
B.1	Perceived MPI, effective MPI and effective POSIX bandwidths for IOR through HDF-5 on Minerva, Sierra and BG/P.	130
B.2	Perceived MPI, effective MPI and effective POSIX bandwidths for IOR through MPI-IO on Minerva, Sierra and BG/P.	131
B.3	Perceived MPI, effective MPI and effective POSIX bandwidths for FLASH-IO through HDF-5 on Minerva, Sierra and BG/P.	131
B.4	Perceived MPI, effective MPI and effective POSIX bandwidths for BT class C on Minerva, Sierra and BG/P.	132
B.5	Perceived MPI, effective MPI and effective POSIX bandwidths for BT class D on Minerva and Sierra.	132
C.1	MPI and POSIX function statistics for FLASH-IO on Minerva.	133
C.2	MPI and POSIX function statistics for FLASH-IO on Sierra 12 to 96 cores.	133
C.3	MPI and POSIX function statistics for FLASH-IO on Sierra 192 to 1536 cores.	133
C.4	MPI and POSIX function statistics for FLASH-IO on BG/P.	134
C.5	FLASH-IO performance on Minerva and Sierra with collective buffering and data sieving optimisation options.	134

E.1	Read and write performance of PLFS through FUSE, the <code>ad_plfs</code> MPI-IO driver and LDPLFS compared to the standard <code>ad_ufs</code> MPI-IO driver on Minerva, using 1 core per node.	137
E.2	Read and write performance of PLFS through FUSE, the <code>ad_plfs</code> MPI-IO driver and LDPLFS compared to the standard <code>ad_ufs</code> MPI-IO driver on Minerva, using 2 cores per node.	138
E.3	Read and write performance of PLFS through FUSE, the <code>ad_plfs</code> MPI-IO driver and LDPLFS compared to the standard <code>ad_ufs</code> MPI-IO driver on Minerva, using 4 cores per node.	139
E.4	Write performance in BT class C for PLFS through the <code>ad_plfs</code> MPI-IO driver and LDPLFS compared to the standard <code>ad_ufs</code> MPI-IO driver on Sierra.	140
E.5	Write performance in BT class D for PLFS through the <code>ad_plfs</code> MPI-IO driver and LDPLFS compared to the standard <code>ad_ufs</code> MPI-IO driver on Sierra.	140
E.6	Write performance in FLASH-IO for PLFS through the <code>ad_plfs</code> MPI-IO driver and LDPLFS compared to the standard <code>ad_ufs</code> MPI-IO driver on Sierra.	140
F.1	Numerical data for Figure 6.2, displaying bandwidth achieved by IOR on 1,024 cores, while varying Lustre stripe size and stripe count.	141
F.2	Numerical data for Figure 6.3, displaying bandwidth per task under contention, along with the idealised values.	141
G.1	Stripe collision statistics for PLFS backend directory running with 16 cores.	142
G.2	Stripe collision statistics for PLFS backend directory running with 32 cores.	142
G.3	Stripe collision statistics for PLFS backend directory running with 64 cores.	142

G.4	Stripe collision statistics for PLFS backend directory running with 128 cores.	143
G.5	Stripe collision statistics for PLFS backend directory running with 256 cores.	143
G.6	Stripe collision statistics for PLFS backend directory running with 512 cores.	143
G.7	Stripe collision statistics for PLFS backend directory running with 1,024 cores.	144
G.8	Stripe collision statistics for PLFS backend directory running with 2,048 cores.	144
G.9	Numeric data for Figure 6.6, showing the performance of IOR through Lustre and PLFS.	144

CHAPTER 1

Introduction

Since the birth of the modern computer, in the early 20th Century, there has been a dramatic shift in how science is performed; where previously countless experiments were performed with varying results and levels of accuracy, now simulations are performed ahead of time, reducing – and in some cases eliminating – the number of experiments that need to be performed. To handle the burden of simulating and predicting the outcome of these experiments, computers have become evermore complex and powerful; the most powerful supercomputer at the time of writing can perform 33 quadrillion (33×10^{15}) floating point operations every second [87], and there is a hope that within the next decade this will be increased to 1 quintillion (10^{18}) operations per second [25, 40].

Achieving this level of performance relies on an enormous amount of parallelism – the world’s fastest supercomputer in November 2013, Tianhe-2, consists of 3,120,000 distinct processing elements operating in parallel [87]. The sheer size of the problems being calculated on machines such as this means that loading data from disk often becomes a burden at scale. Furthermore, the number of components in use in these machines has a serious effect on their reliability, with most production supercomputers experiencing frequent node failures [58, 116, 149]. To combat this, resilience mechanisms are required that often involve writing large amounts of data to persistent storage, such that in the event of a failure, the application can be restarted from a checkpoint, avoiding the need to relaunch the computation from the very beginning.

Unfortunately, the persistent storage available on large parallel systems has not kept pace with the development of microprocessors; checkpointing is becoming a bottleneck in many science applications when executed at extreme scale.

Fiala et al. show that at 100,000 nodes, only 35% of runtime is spent performing computation, with the remaining time spent checkpointing and recovering from failures [46]. As the era of exascale computing approaches, this performance gap is widening further still.

1.1 Motivation

The increasing divergence between compute and I/O performance is making analysing and improving the state of current generation storage systems of utmost importance. Improvements to I/O systems will not only benefit current-day applications but will also help inform the direction that storage must take if exascale computing is to become practically useful. This thesis demonstrates methods for analysing the performance of I/O intensive applications and shows that by making small changes to how parallel libraries are currently used performance can be improved; furthermore, with the correct combination of software libraries and configuration options, performance can be increased by an order of magnitude on present day systems.

This thesis also contains an investigation into one potential solution to poor parallel file system performance. The parallel-log structured file system (PLFS) is reported to be providing huge improvements in write performance [11,103] and this thesis investigates these claims; specifically it is shown that while many of the techniques used in PLFS may prove important on future systems, on many current day systems, PLFS induces a performance penalty at scale.

1.2 Thesis Contributions

The research presented in this thesis makes the following contributions:

- The development and deployment of an I/O tracing library (RIOT) is described in detail. RIOT is a dynamically loadable library that intercepts the function calls made by MPI-based applications and records them for

later analysis. This is demonstrated using industry-standard benchmarks to show how their performance differs between three distinct supercomputers with a variety of I/O backplanes. RIOT allows application developers to visualise how data is written to the file system and identify potential opportunities for optimisation. In particular, through the analysis of an HDF-5 based code, it is shown that by changing some of the low-level configuration options in MPI-IO, a performance improvement of at least $2\times$ can be achieved;

- Using RIOT, the performance of PLFS is analysed on two commodity clusters. The analysis presented in this thesis not only explains why PLFS produces large speed-ups for general users on large file systems but also suggests that there exists a tipping point where PLFS may harm parallel I/O performance beyond a certain number of cores. The burden of installing and using PLFS is also addressed in this thesis, where a simpler, more convenient method of using PLFS is developed. This pre-loadable library, known as LDPLFS, allows application developers and end users to assess the applicability of PLFS to their codes before investing further time and effort into using PLFS natively;
- Building upon previous work [10, 76, 148], the performance of the Lustre-optimised MPI-IO driver is analysed. On the systems operated by the Lawrence Livermore National Laboratory (LLNL), used throughout this thesis, a Lustre-optimised driver (`ad_lustre`) is not available by default, and this is also true of other studies, whereby a potential optimisation is compared against a Lustre file system using the unoptimised UNIX file system MPI-IO driver (`ad_ufs`) [11]. In this thesis, a customised MPI library is built in order to measure the impact of the specialised driver – demonstrating a potential $49\times$ boost in performance. This thesis extends previous works, demonstrating that although the optimal performance is found by using the maximum amount of parallelism available, this may

not be optimal for a system with many I/O intensive applications competing for a shared resource. A number of metrics are presented to aid procurement decisions and explain potential performance deficiencies that may occur;

- The metrics presented to explain the effect of job contention on parallel file systems are adapted and used to explain the performance defects in PLFS at scale, demonstrating that at 4,096 cores each storage target is being contented by 17 tasks in the average case, with some targets experiencing as many as 35 collisions. The equations presented in this thesis will allow scientists to make decisions about whether PLFS will benefit a given application if the scale at which it will be run and the number of file system targets available is known beforehand. At large scale, Lustre with an optimal set of configuration options outperforms PLFS by $5.5\times$, and induces much less contention on the whole file system, thus benefitting the shared file system as a whole.

1.3 Thesis Overview

The remainder of the thesis is structured as follows:

Chapter 2 contains an overview of current work in the field of high performance computing. Specifically, it describes work related to improving I/O and file system performance, with a focus on the methods that can be used to increase the performance of data intensive applications. This chapter also contains a literature review of current work in the fields of performance benchmarking, system profiling and performance modelling, both analytical and simulation-based.

Chapter 3 presents a brief explanation of the hardware and software environments used in this thesis. The chapter begins with a brief introduction to how spinning-disk-based file systems function, from the operation of the single disks

themselves, up to the distributed file systems that bring all the components together. Chapter 3 concludes with an overview of the applications and systems used throughout this thesis.

Chapter 4 describes the development and use of RIOT, an I/O tracing toolkit designed to analyse the usage patterns in parallel MPI-based applications. The overheads associated with using RIOT are studied, showing that the performance impact is negligible, motivating its use in this thesis. RIOT is used to assess the performance of both IBM’s General Parallel File System (GPFS) and the Lustre file system which are commonplace on leading contemporary supercomputers. GPFS on an IBM BlueGene/P is shown to significantly outperform GPFS and Lustre on commodity clusters due to the use of an optimised MPI-IO driver, specialised aggregator nodes and a tiered storage architecture. The performance of applications dependent on the HDF-5 data formatting library is shown to be suboptimal on two of the clusters used throughout this thesis and, through analysis with RIOT, its performance is improved using a more optimal set of MPI hints.

Chapter 5 contains an analysis of PLFS, a virtual file system developed at the Los Alamos National Laboratory (LANL), showing that at mid-scale PLFS achieves a significant performance improvement over the system’s “stock” MPI library. The reasons for this performance improvement are analysed using RIOT, showing that the use of multiple file streams increases the parallelism available to applications. Due to the burden of installing PLFS on shared resources, a rapid deployment option is developed called LDPLFS – a preloadable library that can be used not only with MPI-based applications but also with the standard UNIX tools, where the PLFS FUSE mount is not available. LDPLFS is deployed on two supercomputers, showing that its performance matches that of PLFS through the MPI-IO driver.

Chapter 6 analyses previous works in improving performance on Lustre file systems [9, 10, 76, 148] and expands upon them, showing that although the optimal

configuration produces a $49\times$ performance increase in isolation, the performance increase is nearer $10-12\times$ on a system shared with multiple I/O intensive application. Further, it is shown that using fewer resources has a negligible impact on performance, while freeing up a significant amount of resources. In Chapter 5, performance degradation was observed in PLFS at scale; this chapter analyses why this slowdown occurs. Finally, this chapter presents a number of metrics for assessing the impact of job contention on parallel file systems, and the use of PLFS. These equations could be used to inform purchasing and configuration decisions.

Chapter 7 concludes the thesis, and discusses the implications of this research on future I/O systems. The limitations of the research contained therein are discussed and directions for ongoing and future work are presented.

CHAPTER 2

Performance Analysis and Engineering

Improving computational performance has been a long standing goal of many scientists and mathematicians for thousands of years, even before the advent of the modern computer. Devising more efficient algorithms to solve computational problems can reduce the time taken to reach a solution by many orders of magnitude, meaning calculations relating to natural phenomena can be performed in seconds rather than weeks or months.

The earliest known examples of algorithm optimisation come from Babylonian mathematics [72]. Tablets dating back to around 3000 B.C.E. show that the Babylonians had algorithms that today read very much like early computer programs. These algorithms allowed the Babylonians to efficiently and accurately calculate the results of divisions and square roots, amongst other things.

A more modern example of algorithm optimisation was used during the Manhattan Project at the Los Alamos National Laboratory (LANL). Richard Feynman devised a method for distributing the calculations for the energy released by different designs of the implosion bomb [64]. Through Feynman's use of pipelining, his team of human computers were able to produce the results to 9 calculations in only 3 months, where 3 calculations had previously taken 9 months to produce – representing a $9\times$ speed-up. Distributed computation in this manner is one form of what is now commonly called *parallel computation*.

This chapter summarises: (i) some of the basic concepts and terminology used in parallel computation and high performance computing literature; (ii) some of the principles used to analyse, reason about, and predict computing performance; and finally, (iii) recent advances in performance engineering, with a particular focus on I/O and parallel storage systems.

2.1 Parallel Computation

The first general-purpose computer was the Electronic Numerical Integrator and Computer (ENIAC), built in 1939. The machine was capable of performing between 300 and 500 floating point operations per second (FLOP/s). Due to the prevalence and importance of floating point operations in modern day science applications, the FLOP rate is the standard way in which modern supercomputer performance is assessed.

The era of the *modern* supercomputer began in the 1960s with the release of the CDC 6600. Designed by Seymour Cray for the Control Data Corporation (CDC), the CDC 6600 was the first mainframe computer to separate many of the components, typically found in CPUs of the era, into separate processing units. This resulted in the CPU being able to use a reduced instruction set, simplifying its design, and also allowing operations usually performed by the CPU (such as memory accesses and I/O) to instead be handled by dedicated peripheral processors in parallel. Consequently, the CDC 6600 was approximately three times faster than its predecessor, the IBM 7030, and the machine held the record for the world's fastest computer from 1964 to 1969, performing approximately 1 million floating point operations per second (1 MFLOP/s).

In the 50 years since the CDC 6600, supercomputers have become increasingly more complex. The use of advanced features such as instruction pipelining, branch prediction and SIMD (single-instruction, multiple-data) instruction sets, has led to modern CPUs achieving up to 10 GFLOP/s of computational power per core. A typical CPU now consists of multiple cores (as many as 16 cores on some AMD Opteron CPUs, and many more on some GPUs and specialised processors) and a single CPU can provide as much as four orders of magnitude more performance than the CDC 6600's processor.

As a result of the ever-increasing power of supercomputers, a broad range of applications are now executed on them. Some algorithms are inherently serial, and thus the increase in single core performance has benefitted them. The grow-

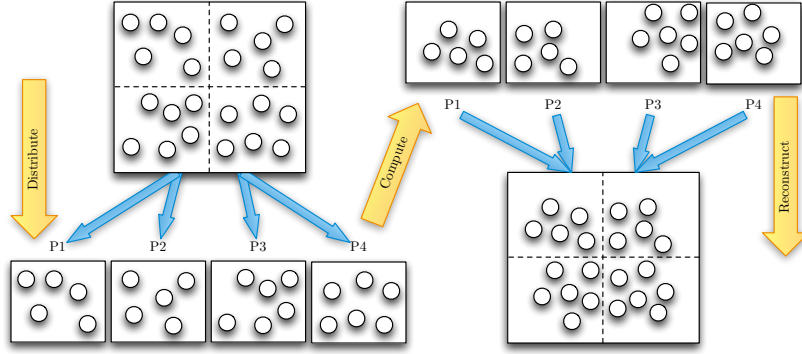


Figure 2.1: An example of the parallelisation of a simple particle simulation between four processors.

ing size of today’s supercomputers also means that many more of these types of application can be executed simultaneously. The increasing core density in modern CPUs is benefitting application that use shared-memory, such as those written using OpenMPI directives [31]. However, this thesis focuses largely on applications using the *data-parallel* paradigm, where an application divides its data across many processors, all working towards a common goal. These applications may use a partitioned global address space (PGAS) model, where the memory is logically partitioned and shared between cooperating processes, or use a message passing model, where messages are explicitly exchanged between cooperating processes.

This thesis focuses on applications using message passing, as they (i) represent a large proportion of the work performed on modern day supercomputers; (ii) make the most use of parallel file systems; and (iii) will benefit most from any optimisations to parallel I/O.

Figure 2.1 represents the division of a particle simulation across four processors. Typically a problem space is divided evenly between cooperating processors, the local problems are solved, and then a communication phase takes place to exchange border information. The computation of the next time step can then commence. After a defined number of time steps, the problem space can be recombined and the result stored.

Because of the significant decrease in runtime when applications are parallelised in this way, supercomputers are now used to investigate a wide variety of problems in both academia and industry. High performance computing is used across a wide variety of domains such as cancer research, weapons design and automotive aerodynamics, as well as investigating astrophysical phenomena such as star formation.

2.2 I/O in Parallel Computing

As supercomputers have grown in compute power, so too have they grown in complexity, size and component count. With the push towards exascale computing (estimated by 2022 at the time of writing [33]), the explosion in machine size will result in an increase in component failures. To calculate the speed of the *Sequoia* supercomputer, the computational benchmark (LINPACK [81]) required multiple execution attempts due to the difficulty of keeping every compute node running for the required 23 hour computation, and this problem is expected to get worse at exascale.

To combat reliability issues, long running scientific simulations now use checkpointing to reduce the impact of a node failure. Periodically, during a time consuming calculation, the system's state is written out to persistent storage so that in the event of a crash, the application can be restarted and computation can be resumed with a minimal loss of data. Furthermore, frequent checkpointing facilitates another important scientific endeavour – visualisation.

With a stored state recorded at set points in computation, scientists can load these checkpoints into a visualisation tool and observe the state of a simulation at various time steps.

2.2.1 Issues in Parallel I/O

Writing checkpoints or visualisation data from a serial application may be relatively trivial but for a parallel application, coordinating the writing or reading

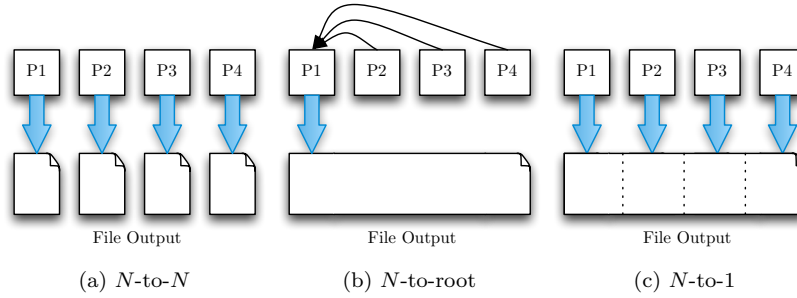


Figure 2.2: The three basic approaches to I/O in parallel applications.

process can be difficult. This has resulted in a number of solutions with various advantages and disadvantages. Figure 2.2 shows three approaches to outputting data in parallel, where (a) all ranks write their own data file; (b) all ranks send their data to one “writer” process; and finally, (c) all ranks write their data to the same file in parallel¹.

While the fastest performance is usually achieved using the approach shown in Figure 2.2(a), this is also the most difficult to manage. If the application is always executed in the same fashion (using exactly the same number of processes) this is the most efficient approach. However, should the problem be run on a differing number of cores (e.g. initially executed on N cores, writing N files, before reloading data from N files, but on M cores), reloading the data becomes computationally expensive and complicated as each process must read sections from multiple different files.

Figure 2.2(b) shows the case where the root process becomes a dedicated writer, writing all of the data to a single file, and redistributing the data in the event of a problem reload. This is the easiest writing fashion to manage but is also the slowest. While the computation is taking advantage of the increased parallelism, the I/O becomes a serialisation point.

The approach taken by most simulation applications is demonstrated in Figure 2.2(c). This approach strikes a good balance between speed and man-

¹Figure 2.2(c) represents a simplified case where each rank is only writing out values from a single shared array. More complicated write patterns (such as data striding) are commonplace.

ageability. This is also the approach most parallel file systems are designed for, and many communication libraries provide convenient APIs for handling data in this manner.

2.2.2 Parallel File Systems

A hard disk drive (HDD) is essentially a serial device; one piece of data can be sent over the connector at any given time. The inner workings of a single HDD and how this has been improved over time will be discussed in Chapter 3, but when multiple parallel threads or simultaneously running applications are using a single storage system, the total performance of the disk will decrease due to the overhead associated with resource contention. On large parallel supercomputers, not only is a single HDD not nearly large enough to handle the required data workloads, but the performance would also decrease to the point of the HDD being practically unusable. To produce a greater quality of service (QoS) across a shared platform, large I/O installations are necessary, using thousands of disks connected in parallel using technologies such as Redundant Array of Independent Disks (RAID) [97] and distributed file systems (DFS).

Distributed File Systems

The I/O backplane of high-performance clusters is generally provided by a DFS. The two most widely used file systems today are IBM's General Parallel File System (GPFS) [115] and the Lustre file system [117], both of which will be discussed in more detail in Chapter 3.

Most DFSs in use today provide parallelism by offering simultaneous access to a large number of file servers within a common namespace – files are divided into blocks and distributed across multiple storage backends. An application running in parallel may then access different parts of a given file without the interactions colliding with each other, as each block may be stored on a different server.

However, the use of a common namespace complicates DFSs – in the Lus-

tree file system, a dedicated server is used to maintain the directory tree and properties of each file, while in GPFS the metadata is distributed across the file servers, complicating some operations but potentially providing higher performance metadata queries.

One precursor to both Lustre and GPFS was the Parallel Virtual File System (PVFS) developed primarily at the Argonne National Laboratory (ANL) [22]. PVFS used the same object-based design [85] that is now common in almost all DFSs and, like Lustre, used a single metadata server to manage the directory tree. However, over time PVFS (and its successor PVFS2) has adopted distributed metadata to decrease the burden on a single server. Likewise, the Ceph file system strikes a balance between Lustre and GPFS by distributing metadata across multiple servers. In Ceph, directory subtrees are mapped to particular servers using a hashing function, though larger directories are mapped across many servers to provide higher performance metadata operations [137].

Hedges et al. suggest that GPFS outperforms Lustre for almost all tasks, except some metadata tasks, where Lustre uses caching to improve performance while GPFS performs a disk flush and read [63]. Furthermore, Logan et al. suggest smaller stripe sizes on a Lustre system lead to better performance [79]. The findings in this thesis and other literature demonstrates that much of the differences in performance can be explained by differing hardware and software configurations [9, 10, 142]. This thesis also suggests that larger stripe sizes may be beneficial on some Lustre file systems at scale.

Although most DFSs provide a POSIX-compliant interface (allowing standard UNIX tools like `cp`, `ls`, etc. to be used), the best performance is often achieved using their own APIs.

Virtual File Systems

In addition to DFSs, a variety of virtual file systems have been developed to improve performance. One approach shown to produce large increases in write bandwidth is the use of so called *log-structured* file systems [104]. When per-

forming write operations, the data is written sequentially to persistent storage regardless of intended file offsets. Writing in this manner reduces the number of expensive seek operations required on I/O systems backed by spinning disks. In order to maintain file coherence, an index is built alongside the data so that it can be reordered when being read. In most cases this offers a large increase in write performance, which benefits checkpointing, but does so at the expense of poor read performance.

In the Zest implementation of a log-structured file system, the data is written in this manner (via the fastest available path) to a temporary staging area that has no read-back capability [94]. This serves as a transition layer, caching data that is later copied to a fully featured file system at a non-critical time.

As well as writing sequentially to the disk, *file partitioning* has also been shown to produce significant I/O improvements. Wang et al. use an I/O profiling tool to guide the transparent partitioning of files written and read by a set of benchmarks [135,136]. Through segmenting the output into several files spread across multiple disks, the number of available file streams is increased, reducing file contention on the storage backplane. Furthermore, file locking incurs a much smaller overhead as each process has access to its own unique file.

The parallel log-structured file system (PLFS) from LANL combines file partitioning and a log-structure to improve I/O bandwidth [11]. In an approach that is transparent to an application, a file access from N processes to 1 file is transformed into an access of N processes to N files. The authors demonstrate speed-ups of between $10\times$ and $100\times$ for write performance. Due to the increased number of file streams, they also report an increased read bandwidth when the data is read back on the same number of nodes used to write the file [103].

With PLFS representing a single file as a directory of files, where each MPI rank creates 2 files (an index file and a data file), there can be an enormous load created on the underlying file system's metadata server. Jun He et al. demonstrate this, suggesting methods for reducing this burden and thus accelerating the performance of PLFS further [62].

While log-structured file systems usually produce a decrease in read performance, the use of file partitioning in PLFS improves read performance to a much greater extent on large I/O systems [103]. PLFS is described in more depth in Chapter 3 and its performance is analysed in Chapter 5.

2.2.3 Parallel I/O Middleware

Writing data in parallel can be a complicated process for programmers; ensuring the output doesn't suffer from race conditions may require explicit offset calculations or file locking semantics. To simplify this process, there are a range of parallel libraries that abstract this complex behaviour away from the application.

Just as the Message Passing Interface (MPI) has become the *de facto* standard library used to abstract data communication from parallel applications, so too has MPI-IO become the preferred method for abstracting parallel I/O [86]. The ROMIO implementation [127] – used by OpenMPI [49], MPICH2 [56] and various other vendor-based MPI solutions [2, 15] – offers a series of potential optimisations, closing the performance gap between *N*-to-*N* and *N*-to-1 file operations.

Within MPI-IO itself there are two features applicable to improving the performance of all parallel file systems. Firstly, *collective buffering* (demonstrated in Figure 2.3) has been shown to yield a significant speed-up, initially on applications writing relatively small amounts of data [92, 126] and more recently on densely packed nodes [142]. These improvements come in the first instance due to larger “buffered” writes that make better use of the available bandwidth and in the second instance due to the aggregation of data to fewer ranks per node, reducing on-node file system contention.

Secondly, *data-sieving* has been shown to be extremely beneficial when using file views to manage interleaved writes within MPI-IO [126]. In order to achieve better utilisation of the file system, a large block of data is read into memory before small changes are made at specific offsets. The data is then written

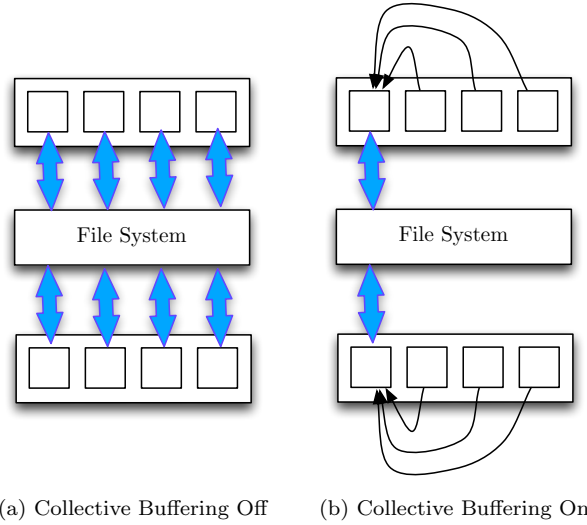


Figure 2.3: An example of two nodes (four ranks per node) writing to a file system with collective buffering off and on.

back to the disk in a single block. This decreases the number of seek and write operations that need to be performed at the expense of locking a larger portion of the file and therefore may benefit sparse writes, where small portions of data may need to be updated [26].

The MPI-IO specification outlines ADIO, an abstract interface for providing custom file system drivers to improve the performance of parallel file systems [125]. On the IBM BlueGene/L (and subsequent generations), a custom driver is provided for GPFS (`ad_bg1`) [2]. As these drivers are aware of the file system’s APIs, they do not rely on unoptimised POSIX-compliant alternatives. As is demonstrated later in this thesis, performance can be boosted significantly through using file system specific drivers.

For the Lustre file system, the `ad_lustre` driver is provided in the standard ROMIO distribution [35, 36]. Using the driver allows an application developer to specify additional options to customise the file layout at runtime, potentially increasing the parallelism available [9, 10, 100, 101].

In addition to drivers within the MPI-IO framework there are middleware layers that exist between the applications and parallel communication libraries

designed to standardise the I/O in scientific applications. NetCDF [106] and Parallel NetCDF [75] exist for this purpose, with Parallel NetCDF making use of the MPI-IO library to provide parallel and improved performance.

More commonly, the hierarchical data format (HDF-5) is used to write data to disk for checkpointing or analysis purposes [73]. The library can be compiled and can operate with the MPI library to allow parallel access to a common data file; in this way the library can make use of optimisations in MPI to increase performance [66, 146]. Additionally, PLFS has been demonstrated to improve the performance of HDF-5 based applications by Mehta et al., dividing a single HDF-5 output file into a data layout that is more optimal for the underlying file system [84].

In this thesis, two applications that make use of HDF-5 are analysed, demonstrating the shortcomings that may exist in the library's default configuration, while presenting opportunities for optimising performance.

2.3 Performance Engineering Methodologies

In high performance computing parlance, *performance engineering* is the collection of processes by which an application's or computing system's performance is measured, predicted and optimised. With supercomputers typically costing anywhere between £1.4 million (approximate cost of Minerva, the University of Warwick operated supercomputer used throughout this thesis) and £750 million (approximate cost of K-computer, the 10 PFLOP/s supercomputer installed at the RIKEN Advanced Institute for Computation Science in Kobe, Japan), understanding the potential performance and utility of these machines ahead of procurement is becoming significantly more important [60]. In addition to making sense of the performance of a parallel machine, it is also important to understand the performance of the applications that are expected to run on these systems.

Further to system procurement, performance engineers also require tools to

assess the current performance of their applications in order to understand why they perform as they do. With this data, optimisations can be made, alongside predictions about how hardware or software changes may affect the performance of their applications [32, 98].

2.3.1 Benchmarking

The most common way to assess a new computing architecture or parallel file system is through the use of benchmarking. There exist multiple benchmarks specifically designed for the assessment of supercomputers and many of these benchmark suites form the basis of various performance rankings [6, 28, 39, 81]. For example, the LINPACK benchmark is a linear solver code that produces a performance number (in FLOP/s) that is used to rank the most commonly cited list of the fastest supercomputers, namely the TOP500 list [87].

For the purpose of procurement, running LINPACK on a small test machine and extrapolating the performance forward can produce an approximation of the parallel performance of a much larger, similarly architected machine (since LINPACK scales almost linearly [39]). Additionally, benchmarks such as STREAM [83] and SKaMPI [68] exist to assess the performance of memory and communication subsystems.

The aforementioned benchmarks all target particular facets of parallel machines that are particularly important to performing computation. For data-driven workloads, there are a number of benchmarks specifically designed to assess the performance of the parallel file systems attached to these systems. Notable tools in this area include the IOR [121] and IOBench [139] parallel benchmarking applications. While these tools provide a good indication of *potential* performance, much like LINPACK, they are rarely indicative of the true behaviour of *production* codes. For this reason, a number of mini-application benchmarks have been created that extract file read/write behaviour from larger codes to ensure a more accurate representation of an application's I/O operations. Examples include the Block Tridiagonal (BT) solver application from the

NAS Parallel Benchmark (NPB) Suite [7, 8] and the FLASH-IO [47, 109, 155] benchmark from the University of Chicago – both of which are employed later in this thesis.

2.3.2 System Monitoring and Profiling

While benchmarks may provide a measure of file system performance, their use in diagnosing problem areas or identifying optimisation opportunities within large codes is limited. For this activity, monitoring or profiling tools are required to either sample the system’s state or record the system calls of parallel codes in real-time.

The *gprof* tool is often used in code optimisation to identify particular functions that consume a large about of an application’s runtime [55]. For parallel applications this task is complicated, as the program is spread across a wide number of processes; a parallel profiler is therefore required for these applications. For Intel architectures, the VTune application can inform an engineer how the CPU is being used, how the cache is being used and much more [69]. Oracle Solaris Studio (formally, Sun Studio) consists of high performance compilers in addition to a collection of performance analysis tools [27].

For parallel applications, there are a range of tools specifically designed to monitor and record data relating to inter-process communications. Notable tools in this area include the Integrated Performance Monitoring (IPM) suite [48] from the Lawrence Berkley National Laboratory (LBNL), Vampir [90] from TU Dresden, Scalasca [50] from the Jülich Supercomputing Centre, Tau [122] from the University of Oregon and the MPI profiling interface (PMPI) [38]. Each of these profiling tools record interactions with the MPI library, and thus produce large amounts data useful for identifying communication patterns and performance bottlenecks in parallel applications. Further, both Scalasca and Tau can generate additional data relating to performance using function call-stack traversal and hardware performance counters.

For monitoring I/O performance the tools `iostat` and `iostat` both monitor

a single workstation and record a wide range of statistics ranging from the I/O busy time to the CPU utilisation [74]. `iotop` is able to provide statistics relevant to a particular application, but this data is not specific to a particular file system mount point. `iostat` can provide more detail that can be targeted to a particular file system, but does not provide application-specific information. These two tools are targeted at single workstations, but there are many distributed alternatives, including Collectl [118] and Ganglia [82].

Collectl and Ganglia both operate using a daemon process running on each compute node and therefore require some administrative privileges to install and operate correctly. Data about the system's state is sampled and stored in a database; the frequency of sampling therefore dictates the overhead incurred on each node. The I/O statistics generated by the tools focus only on low-level POSIX system calls and the load on the I/O backend and therefore the data will include system calls made by other running services and applications. For more specific information regarding the I/O performance of parallel science codes many large multi-science HPC laboratories (e.g. ANL, LBNL) have developed alternative tools.

Using function interpositioning, where a library is transparently inserted into the library stack to overload common functions, tools such as Darshan [21] and IPM [48] intercept the POSIX and MPI file operations. Darshan has been designed to record file accesses over a prolonged period of time, ensuring that each interaction with the file system is captured during the course of a mixed workload. The aim of the Darshan project is to monitor I/O activity for a substantial amount of time on a production BG/P machine in order to guide developers and administrators in tuning the I/O backplanes used by large machines [21].

Similarly, IPM uses an interposition layer to catch all calls between the application and the file system [48]. This trace data is then analysed in order to highlight any performance deficiencies that exist in the application or middleware. Based on this analysis, the authors are able to optimise two applications, achieving a 4× improvement in I/O performance.

ScalaTrace [93] and its I/O-based equivalent ScalaIOTrace [134] have similarly been used to record and analyse the communication and I/O behaviours of science codes. Using the MPI traces collected by ScalaTrace, the authors have demonstrated the ability to auto-generate skeleton applications in order to obfuscate potentially sensitive code for the purpose of benchmarking differing communication strategies and interconnects [34]. Their success in producing applications representative of the communication behaviours of science codes suggests that a similar methodology could be used for building I/O benchmarks.

2.3.3 Analytical Modelling

Performance modelling and simulation have been previously used to predict the compute performance of various science codes at varying scales on hypothetical supercomputers. Analytical models (predominantly based on the LogP [30] and LogGP [3] models) have been heavily used to analyse the scaling behaviour of hydrodynamic [32] and wavefront codes [60, 89], as well as many other classes of applications [13, 16, 51, 71].

Modelling the performance of a single-disk file system may be simple for certain configurations, where all writes are of a fixed size, large enough such that caching effects do not skew performance. More complex configurations or usage patterns complicate matters, with issues such as head switches and head seeks changing the performance characteristics.

Ruemmler and Wilkes present an analytical model for head seeks, in which small seeks are handled differently to larger seeks (where the head has the opportunity to reach its maximum speed and therefore coast for a period) [111]. Further, they demonstrate a simulator using analytical models for various aspects of a physical hard disk drive, but use some simulation-based modelling to produce a complete disk model [111]. Shriver et al. produce a complete analytical behaviour model for a hard disk drive, taking into account a simple readahead cache as well as request reordering [123]. Probabilistic functions are used throughout to model cache hits and misses. The culmination of the work is

a model that is within 5% of the observed data for some workload traces, but decreases in accuracy for large multi-user systems with many parallel applications reading from and writing to the file system.

Work has also been conducted into building an analytical model of a parallel file system. Zhao et al. present a performance model for the Lustre file system, demonstrating an average model error of between 17% and 28%, thus illustrating the difficulty of modelling complex parallel I/O systems with a large number of components [152].

While analytical models can produce near instant answers to some performance modelling problems, when faced with heavy machine or file system contention, analytical models fail to produce accurate answers [98]; for these problems, simulation is often required.

2.3.4 Simulation-based Modelling

Two simulation platforms have been developed recently at Sandia National Laboratories (SNL) and the University of Warwick. The Structural Simulation Toolkit (SST), from SNL, provides a framework for both macro-level and micro-level simulation of parallel science applications, simulating codes at an abstract level (predicting MPI behaviours and approximate function timings), as well as at a micro-instruction level [107]. Similarly, the Warwick Performance Prediction (WARPP) toolkit simulates parallel science codes at macro-level, and includes simulation parameters to introduce network contention (through the use of a Gaussian distribution of background network load) [59, 61].

While WARPP only attempts to simulate computation and communication behaviour, SST can also predict I/O performance using an optional plugin to simulate a single hard disk (using DiskSim [14]). However, the module is not included by default and is currently not capable of simulating an entire parallel file system. Simulation of an HDD using DiskSim relies on the target disk being benchmarked using the DIXtrac application, which determines the values for “over 100 performance-critical parameters” [113, 114], including the disk’s data

layout, its seek profile and various disk cache timing parameters; however, much of this feature extraction relies on features of the SCSI interface that are not applicable to modern HDDs. For newer disks, with more complex data layouts, geometry extraction relies on some benchmarking and guesswork.

Specifically, Gim et al. use an angular prediction algorithm, along with a host of other metrics to determine many of these parameters [52,54]. From their data, they can predict the data layout of the disk. Where DIXtrac currently takes up to 24 hours to fully characterise a disk, Gim et al. demonstrate similar accuracy (on newer disks) within an hour [54].

Additional work into disk simulation has been done by both IBM and Hewlett-Packard (HP) Laboratories. Hsu et al. use a trace driven simulation to analyse the performance gains of various I/O optimisations and disk improvements [67]. They assess the benefits of read caching, prefetching and write buffering, demonstrating their benefits to improving I/O performance. Likewise, Ruemmler and Wilkes assess the impact of disk caching using a simulation, demonstrating a large error in predictions for small operations when the cache is not modelled, highlighting the importance of disk cache modelling [111].

Early disk caches (typically less than 2 MB in size) would partition their available storage into equally sized blocks to allow multiple simultaneous read operations to use the cache. Modern hard disks do not have this same restriction, instead partitioning the cache according to some heuristic. Suh et al. demonstrate, using a simulator, that the disk's cache hit-ratio can be improved by using an online algorithm to dynamically partition the cache [124]. Similarly, Zhu et al. demonstrate the benefit of both read and write caching on sequential workloads, but conclude that there is very little benefit when there are more concurrent workloads than cache segments [154].

Thekkath et al. develop a “scaffold” interface in order to allow them to use a real file system module to simulate performance [129]. Their scaffolding simulator mimics many of the operations that would otherwise be performed by the kernel in order to bypass writing to physical media, instead directing data

towards a disk model.

Simulating parallel file systems is much more difficult, instead requiring the simulation of both a shared metadata target, as well as multiple data targets. Molina-Estolano et al. have developed IMPIOUS [88], a trace-driven parallel file system simulator that attempts to mimic a storage system using PanFS [91]. Although their absolute results are out by an order of magnitude, the trend-line of their results is similar to the true performance.

The CODES storage system simulator has been developed by Liu et al. to predict the performance of a large PVFS2 installation at ANL [77]. They use their model to predict the benefit of burst-buffer solid state drives (SSD) within their installation, concluding that performance may be greatly improved if burst buffer disks were deployed more widely [78].

Finally, Carns et al. use a simulator of PVFS2 in order to demonstrate the inefficiencies in server-to-server communication, used to maintain file consistency [23]. They modify the algorithms used by PVFS2 and demonstrate speed-ups in file creation, file removal and file stat operations.

2.4 Summary

Parallel computers are forever changing, and achieving optimal performance is becoming increasingly difficult as the technology evolves. From its humble beginnings in the laboratories at LANL – using human computers to distribute complex equations – to the current generation billion dollar parallel behemoths, supercomputing has changed how science is performed. In this chapter a survey of current research in HPC has been presented.

Of particular interest, the work performed by Carns et al. [21] and Furlinger et al. [48] inspires much of the work in Chapter 4. Both Darshan and IPM perform similar tasks to the tool described in this thesis, however much less focus is put on associating the MPI library function calls with the underlying POSIX operations that commit data to the file system.

The work of Bent et al. [11] in developing PLFS demonstrates the current divergence between how applications perform I/O and how file systems expect I/O to be performed. In Chapter 5, the performance gains reported in the PLFS literature [11,62,84,103] are investigated, demonstrating that there is still progress to be made in achieving the best performance on current-generation parallel file systems.

Finally this thesis analyses the work of Behzad et al. [10], Lind [76] and You et al. [148] to show that current generation file systems are often better than reported, though this thesis demonstrates that performance often suffers under contention. The work in this thesis also ties the issues associated with file system contention back to PLFS, demonstrating that PLFS has a similar effect to contended jobs when applications are run at large scale.

CHAPTER 3

Hardware and Software Overview

Throughout the work contained in this thesis many different hardware and software systems have been used. This chapter provides a basic overview of how each device works and how various parallel files systems are structured. Finally, the systems and applications used for the experiments in this thesis are summarised.

3.1 Hard Disk Drive

While solid state drives (SSD) are decreasing in cost and improving in performance, mechanical disk drives still dominate on large HPC installations. The adoption of SSD drives is beginning to pick up pace, with the drives already being used in tiered storage systems as burst-buffers (storing recently accessed data and writing to mechanical drives at a later time) [78]. However, in order to understand the performance of current generation parallel storage systems, the effects of mechanical disks must be considered in order to understand the performance of I/O in a multi-user system.

3.1.1 Disk Drive Mechanics

Figure 3.1 shows the basic internal layout of a standard spinning disk. Data is stored on the platters by magnetising a thin film of ferromagnetic material; depending on the magnetic polarity, a particular space on the disk may represent either a 1 or a 0. The disk platters (which may be stacked) can hold data on both sides and the platter assembly rotates at a constant speed. Disks used in laptops and desktop computers typically spin at either 5,400 or 7,200 revolutions per

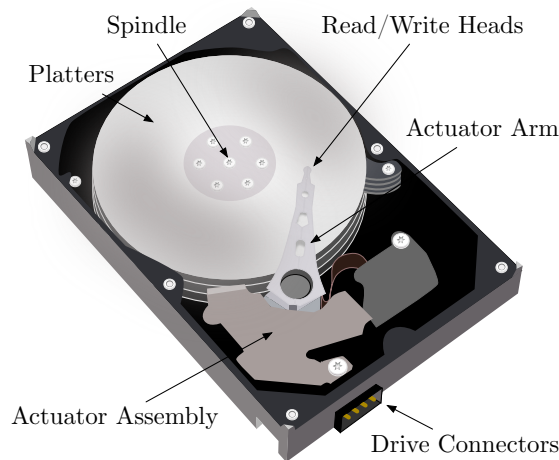


Figure 3.1: Basic internal structure of a hard disk drive^a.

^aImage includes resources from: <http://openclipart.org/detail/28678>

minute (RPM); server systems usually make use of disks that run at 7,200 RPM, 10,000 RPM or even 15,000 RPM.

Data is arranged on the disk platters in concentric circles and the “first” track on a platter is always the outermost. In order to read/write data from/to the tracks, the read/write head is moved over a particular track by the actuator mechanism. The disk controller then enables one of the read/write heads at a time in order to read/write data from/to a specific location.

3.1.2 Data Layout

The data on hard disk drives (HDD) was originally addressed using a method known as cylinder-head-sector (CHS). First the actuator would move the read/write head to the correct cylinder (where a cylinder is the set of tracks on each platter that are equidistant from the spindle), the correct read/write head would be activated and when the particular sector (where a sector is a 512-bit block of data) required was under the read/write head, data would be accessed. However, using a disk in this way wasted a lot of the potential area of the disk platters, as the data density decreases as data is stored further away from the spindle. In addition to this, because of the CHS addressing standard, disks

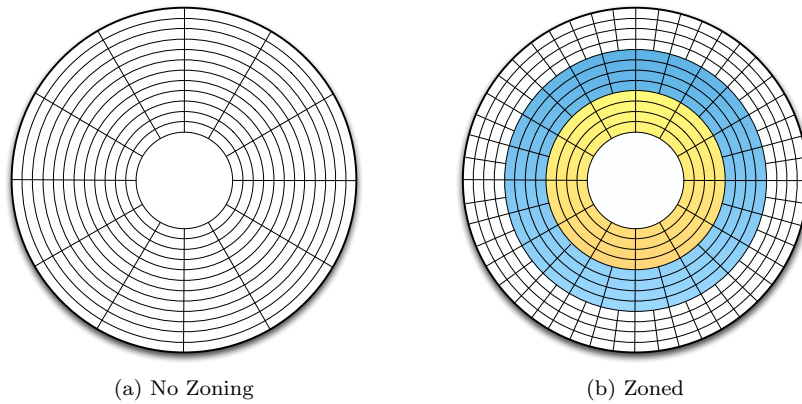


Figure 3.2: Data layout on a disk with no zoning and three zones of increasing density.

were constrained to 65,536 cylinders, 16 heads and 63 sectors per track; giving a total of approximately 4 GB of maximum storage capacity (the limit on older hardware where only 1,024 cylinders are allowed may be as low as 63 MB).

To overcome these limits, modern disk systems now use logical block addressing (LBA), though CHS addressing systems are still present in most BIOS systems for legacy support. In LBA systems, the disk controller takes an address and converts this to a physical address transparently to the user. Because of this, disks can use much more complex data layout schemes.

As the outermost track of a platter is much longer than the innermost track, it can store much more data (though storing data at the same density complicates the read/write heads behaviour and the disk controller). To simplify things, modern disks strike a balance between the complexity of storing all data at the same density and the simplicity of storing all data at the same data rate; the disk is partitioned into multiple zones where each zone has a different data rate. Modern disks typically use between 10 and 30 zones and the complexity of *zoned-bit recording* (ZBR) is handled by the disk drive's internal disk controller [150].

Figure 3.2 illustrates how data may be laid out (a) with a constant data rate, and (b) with three zones. Since the spindle speed of modern disks is constant,

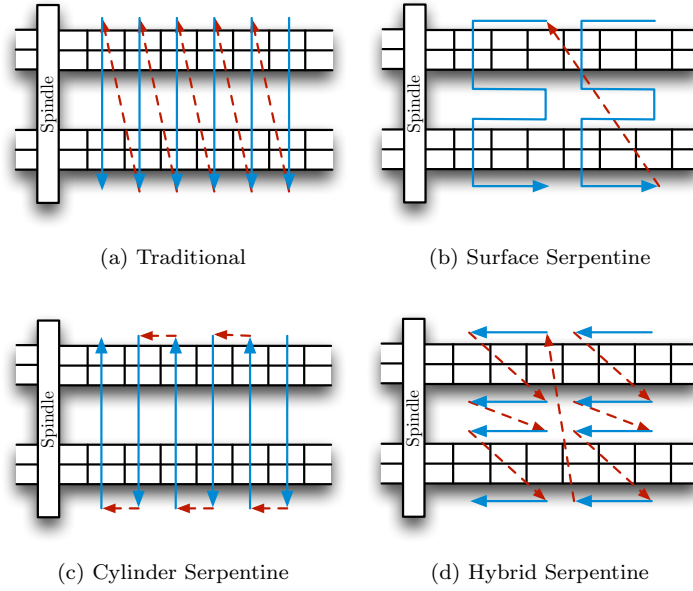


Figure 3.3: Four examples of serpentine sector mapping.

towards the outer edge of a platter more physical space passes under the read head; thus, more data can be stored and read/written in a single rotation of the disk. In Figure 3.2(b), the white zone stores data at double the data rate of the blue zone, which itself has twice the data rate of the yellow zone. As the data rate increases, the available bandwidth to and from the disk increases, thus ZBR not only increases the available capacity, it additionally provides increased performance on the outer tracks. Due to manufacturing processes, translating an LBA address to a physical address is complicated further by the use of differing zone data rates used by each platter; the data rate of a given zone on one surface may not be the same as the corresponding zone on any other surface.

In addition to the use of ZBR to increase the performance of a disk, modern disks use a serpentine pattern for data layout [53, 110, 150]. With modern advances in engineering, it may be more efficient in HDDs to switch to the next track on a disk than to switch head and start reading from another platter (this is due to a head switch requiring an adjustment to the head as well as some

“settling” time). For this reason, a serpentine layout may be used to reduce either the frequency of head switches or track-to-track switches. Figure 3.3 illustrates four common sector mapping schemes. The choice of which layout is used is based on the mechanical aspects of a particular disk; a disk with a high overhead for head switching but a low overhead for track-to-track switches may use of the layout in Figure 3.3(b), whereas a disk with a low overhead for head switches may use the scheme in Figure 3.3(c).

3.1.3 Disk Controller

In order to address the speed divergence between main memory and HDDs (many GB/s for memory, compared to a maximum of ≈ 100 MB/s for HDDs), high speed cache memory is used by the disk controller to buffer data. The disk cache is able to partially nullify the effect of the mechanical operations performed by the disk for largely sequential workloads. Variations of the following cache policies are found on most hard disk drives:

Read-ahead

Data that is sequential to the current request is read and stored in cache as it may be used shortly.

Read-behind

Data prior to the current request is stored in cache as the platters rotate to the correct position as this data may be then be accessed later at no cost, as the head was passing over the data anyway.

Write-through

Data is written to the cache and also to the disk simultaneously. Written data remains in cache as it may be read or updated later.

Write-back

Data is stored only in cache until the cache segment is about to be modified or replaced, at which point the data is committed to disk.

Modern disk caches usually provide between 8 and 32 MB of cache that can be used for either read or write operations. Simple caches on much older disks would partition the small amount of cache memory available into a number of preset cache blocks of a fixed size; a least-recently used (LRU) algorithm would be used to keep the most important data in cache, and write out or overwrite old data. Today, disk caches implement much more complex algorithms using a combination of LRU, least-frequently used and other cache replacement strategies.

Additionally, cache blocks are also dynamically sized and allocated by the disk controller [124, 130]. This added complexity allows much larger sequential reads to be improved when using a large cache. Small reads on a multi-user system can also benefit, as many more smaller cache blocks can be created to deal with each request.

The Serial-ATA (SATA) protocol in 2003 introduced native command queuing (NCQ), allowing disks to re-order operations in a command queue in order to reduce the number of rotations required, thus decreasing the time taken to fulfil queued requests. Figure 3.4 shows how four requests can be reordered, using the elevator algorithm, to decrease the time taken to service the requests [140]; servicing the third request while the disk is spinning to the second request eliminates the need for an extra rotation.

Disk manufacturers also add additional proprietary features to modern hard disks. Due to the commercially sensitive nature of many of these optimisations, it is becoming increasingly difficult to understand the performance of spinning disks, thus any simulations and models must make many simplifying assumptions to provide a generalised model of an HDD.

3.1.4 Redundant Array of Independent Disks

Modern enterprise-class HDDs have a mean time between failures (MTBF) of between 1,200,000 and 2,000,000 hours, representing an annualised failure rate (AFR) of between 0.73% and 0.44% (for the Seagate Constellation ES SAS

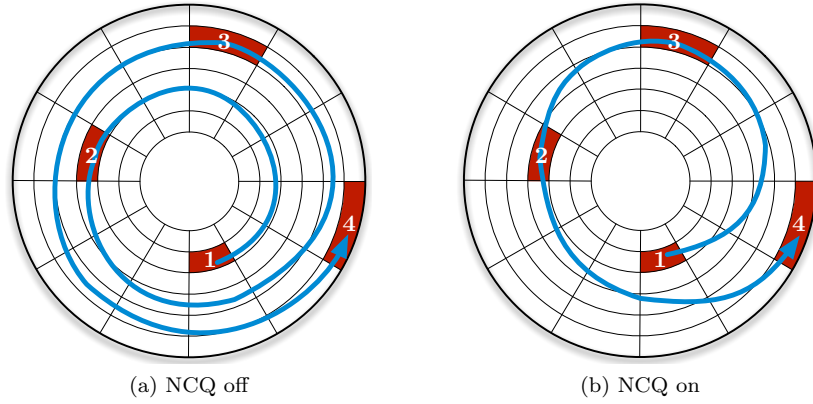


Figure 3.4: An example of four requests fulfilled in-order without NCQ and out of order with NCQ.

HDD, and the Seagate Enterprise Performance 15K HDD, respectively). In a system with thousands of disks, the probability that one of those disks will fail within a year becomes large. To overcome potential drive losses redundancy is required, where some data is duplicated to another disk, such that in the event of a failure, no data is lost.

Patterson, Gibson and Katz outline the first five configurations for redundant arrays of independent disks (RAID)¹ [97]. The first level, RAID-1, allows numerous disks to be connected in parallel, with each pair of disks creating a duplicate of each block simultaneously. This potentially provides better read performance (as each simultaneous read can be processed in parallel), but at the expense of poor write performance, as data duplication requires each disk to synchronise. To reduce the cost of data duplication, RAID-2 uses a Hamming code to provide error correction, while all subsequent RAID levels use parity checking. RAID-3 uses a single dedicated parity disk, while data is striped at a byte level. RAID-4 level is similar to RAID-3, except block-level striping is used with a dedicated parity disk. In the event of a failure, a new disk can be swapped into the RAID system and rebuilt either by recalculating the parity, or by using the present parity data to reconstruct the original data.

¹Initially redundant array of *inexpensive* disks, this has been retroactively changed to be *independent* disks.

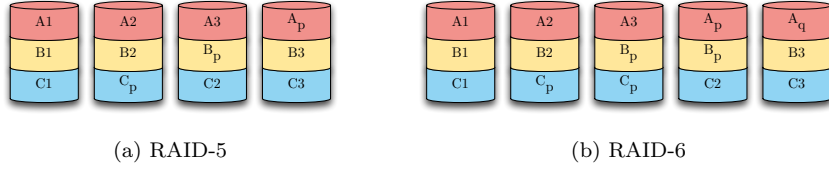


Figure 3.5: Two common RAID data distribution schemes.

RAID-5 and RAID-6 are the most commonly used “standard” levels of RAID at the time of writing, with block-level striping of data and distributed parity. Figure 3.5 shows how data is structured on both RAID-5 and RAID-6, where the parity data is distributed. RAID-5 can survive a single disk failure, while RAID-6 can survive two, as it provides double distributed parity. Due to the ability to access disks in parallel, RAID-5 and RAID-6 also offer potential performance boosts. Specifically, RAID-5 allows read and write performance of $(n - 1)x$, while RAID-6 provides $(n - 2)x$, where n is the number of disks and x is the performance of a single disk.

Additionally, there is RAID-0, where there is no fault tolerance but data is striped to provide performance equivalent to nx . On HPC systems, metadata storage targets often employ RAID-1+0, where data is both striped and mirrored to provide additional resilience and performance.

3.2 File Systems

When writing to an HDD, performance is not only dependent upon the disk drive in use; the performance of the disk is also dictated by how the operating system interacts with the hardware. This is largely controlled by the file system in use.

3.2.1 The Extended File System

The extended file system (ext) was the first file system created specifically for the Linux kernel and was inspired by the UNIX file system (UFS). The file system

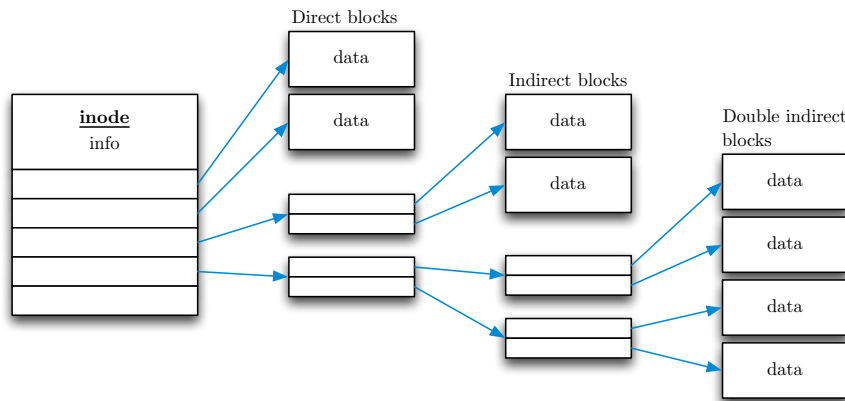


Figure 3.6: Structure of an ext2 inode block.

was superseded almost immediately by the second extended file system (ext2) which used similar metadata structures but remedied many of the limitations of ext. The third iteration of the extended file system (ext3) was developed 8 years later and made use of a journal to improve its reliability and performance. The successor to ext3 was released in 2008 and provided further improvements to performance, mostly in file system checking.

The Second Extended File System

In each of the ext file systems, files are represented by *inodes*, where an inode is a structure that contains all of the information about a file except its filename and the data itself [20]. The POSIX inode description lists the following attributes (amongst others that are not listed below):

- Size of the file in bytes.
- User ID and group ID of the file owner.
- File access modes.
- Creation, last access and last modification times.
- Number of file blocks.
- Pointers to the disk blocks containing the file's data.

Under ext2, the block size for the file system may be 1, 2 or 4 KB and is typically the highest of the three (4 KB). Each inode has a storage area that contains 15 pointers; the first 12 point directly to file data blocks, the next points to an indirect block (which contains pointers to file data blocks), the next points to a double indirect block (which contains pointers to indirect blocks) and the final points to a trebly indirect block. Figure 3.6 outlines the structure of an ext2 inode.

When allocating space for the data blocks of a file, ext2 attempts to preallocate 8 blocks upon file creation; this helps to prevent file fragmentation (where a file's data is not stored contiguously on the physical medium). Additionally, ext2 attempts to place new file blocks as close as possible to the data of other files in the same directory.

The Third Extended File System

The ext3 file system is an extension of ext2 with the addition of a journal (and some other additional features that will not be discussed in this thesis) [133]. The journal on ext3 is essentially a file that is configurable in size and location that stores transactions to the disk. The journal can be configured to work in one of three ways:

Journal

Both metadata and file contents are written into the journal before being committed to the main file system.

Ordered

Only metadata is written to the journal. Data is not, but it is guaranteed that the data will have been written to the file system before the entry is marked as committed in the journal.

Writeback

Only metadata is written to the journal, however the journal entry may be marked as committed before the data has been written to the file system.

The journal may be stored on the file system it is journalling, or it may be stored on another disk or in memory. The advantage of the journal is that in the event of a file system crash, it can be replayed in order to repair the file system and bring the storage back online much more quickly. However, as metadata is usually written to the disk twice (and in some cases the file data may also be written twice), it can have implications for file system performance.

In ext3, the default behaviour is to use *ordered* journalling where only metadata entries are written to the journal, which behaves like a circular log of transactions. At a particular interval, or when the journal reaches its maximum size, the entries are committed to the file system.

The Fourth Extended File System

The fourth extended file system (ext4) is again largely feature compatible with ext2 and ext3 but contains a series of extensions that were originally designed for use by the Lustre file system [117]. The file system allows preallocation of on-disk space of up to 128 MB, allowing applications to reserve large contiguous storage spaces for improved performance. Additionally, the allocation of disk space can be delayed until data is flushed to the file system; this allows a more efficient allocation that may reduce fragmentation and improve performance by using the actual file size for the allocation [133].

In contrast to the previous iterations of the extended file systems, ext4 makes use of file extents instead of traditional block mapping. Rather than using the inode block mapping scheme described above, ext4 inodes can instead store four extents (where an extent is a block of up to 128 MB of contiguous storage). If more than four extents are required for a file, an HTree (a tree data structure specialised for directory indexing) is used to store the additional extents.

3.2.2 The Sun Network File System

The Sun Network File System (NFS) is perhaps the most well known and most widely used file system that provides a unified directory structure to a collection

of clients over a network [112]. It uses Sun's Remote Procedure Call (RPC) protocol in order to allow clients to issue file system commands across the network.

The first public release of NFS (NFSv2) operated using UDP only and therefore provided only a stateless protocol; operations such as a write had to be completed synchronously, whereby the server would have to perform the complete operation before it could return a result to the client. Stateless operation also restricted the use of file locking, and this therefore had to be implemented outside of the core protocol.

Although support for TCP was added in NFSv3, a stateful protocol was not added until NFSv4. Asynchronous write operations were added to NFSv3 in order to provide greater write performance.

3.3 Distributed File Systems

While NFS provides a unified storage solution for many connected nodes, the performance of the file system does not scale with the size of the network. For this reason, distributed file systems (DFS) exist to provide a unified storage space across a network of servers. As a DFS is spread across multiple servers (allowing parallelised access without interprocess interference) it generally provides a greater quality of service (QoS) than NFS. The IBIS file system [131], developed in 1985, was one of the first DFSs where the file system was spread across all nodes of the network, allowing all nodes to transparently access any file regardless of whether the file was stored locally or remotely. Modern DFSs now provide dedicated storage servers, each themselves containing high performance storage backends (using techniques such as RAID).

In most modern DFSs, there are four components. This thesis adopts the naming convention of the Lustre file system; different file systems may use alternate terms but the functions they provide are largely equivalent.

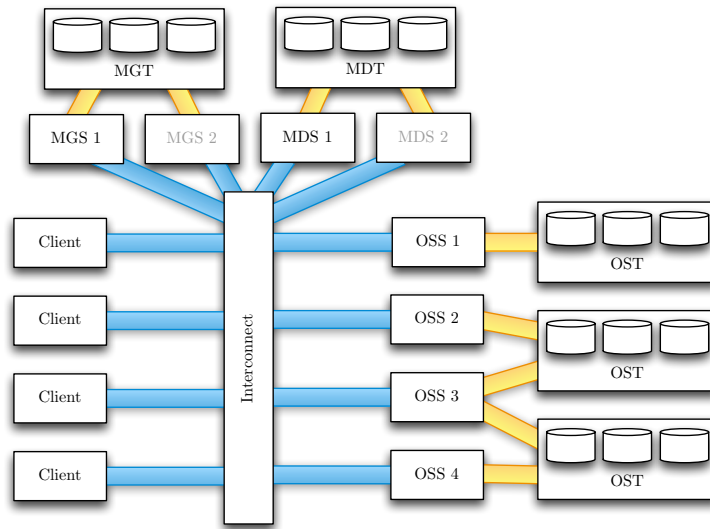


Figure 3.7: An example Lustre configuration with four OSSs and a fail-over MGS and MDS.

Object Storage Targets (OST)

HDDs are usually grouped using RAID (to improve performance and provide some redundancy), these are then referred to as *Object Storage Targets*. The OSTs are used to store the stripe data blocks that make up each file.

Object Storage Servers (OSS)

One or more OSTs are connected to one or more *Object Storage Servers*. The OSSs are directly responsible for reading and writing file data from and to the OSTs.

Metadata Server (MDS)

Metadata (such as the directory tree, file permissions and file block locations) is either stored on a dedicated *Metadata Server* or is stored on the OSSs (as in GPFS). The MDS is used by the clients to get file information and file structure, such that they can access the file stripes stored on the OSTs.

Management Server (MGS)

Finally, there are usually one or two *Management Servers* holding the server configurations.

The parallel file systems used throughout this thesis differ in the number of OSTs and OSSs in use, as well as the use of a dedicated MDS for the Lustre file system used, and distributed metadata on the GPFS installations used. Other parallel file systems, such as Ceph [137], make use of multiple MDSs in order to improve the performance of metadata updates, which are often identified as a bottleneck in some large, heavily loaded Lustre installations [1].

3.3.1 The Lustre File System

The Lustre file system is used by many of the world's fastest and largest computers, with up to 66 of the top 100 HPC systems using Lustre in 2010². The basic architecture of a Lustre file system is shown in Figure 3.7. Although Lustre (up to version 2.4) uses only a single MDS, a fail-over MDS and MGS can be present. Additionally, as shown in Figure 3.7, multiple OSSs can be connected to common OSTs and this will again provide some fail-over capability.

Much like previous DFSs, Lustre makes use of file striping to allow load to be distributed across a number of service nodes. The size and width of each stripe (where the width is the number of servers over which to stripe) can be configured on a per-file or per-directory basis. The Lustre installation used in this thesis stripes across 2 OSTs with each stripe being 1 MB in size in its default configuration. The `lfs` command can be used to view and modify these settings.

When writing to a Lustre system, the server used for the first stripe is randomised in order to provide some load balancing between different clients. From this point onwards, the data is striped across a number of servers based on the configured stripe width.

²<http://opensfs.org/wp-content/uploads/2011/11/Rock-Hard1.pdf>

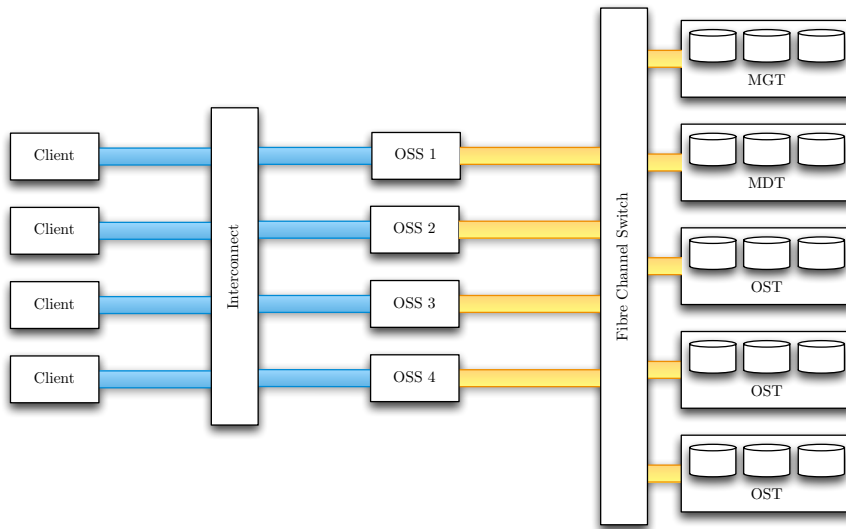


Figure 3.8: An example of a GPFS setup with four OSSs connected via a high performance switch to three targets and separate management and metadata targets.

To maintain consistency and allow correct concurrent access to the DFS, Lustre makes use of a distributed lock manager. Each OSS maintains its own file locks and so if two processes attempt to access the same chunk of a file, the OSS will only grant a lock to one of the clients (unless both accesses are read requests).

3.3.2 IBM's General Parallel File System

The General Parallel File System (GPFS) from IBM operates similarly to Lustre; large files are distributed across multiple storage targets using stripes. However, GPFS differs from Lustre in that all OSSs are connected to all OSTs and MDTs, usually through a fibre channel switch. This provides additional resilience in that many more OSSs can fail before the file system must go offline. Figure 3.8 demonstrates an example GPFS configuration. Although it is possible to store metadata on the same disks as file data, many installations (including the configuration in use at the University of Warwick at the time of writing) make use of dedicated higher performance data targets.

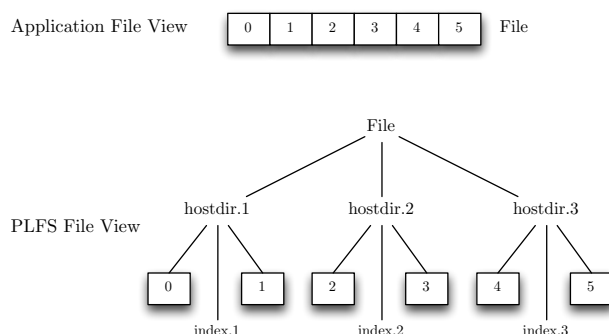


Figure 3.9: An application’s view of a file and the underlying PLFS container structure.

On GPFS, metadata is maintained by all servers, potentially providing better performance for metadata intensive workloads. As shown by Hedges et al., the file creation rate on GPFS is much higher than on a Lustre system, provided that the files are being created in distinct directories; the use of fine grained directory locking in GPFS makes file creation slower in the same directory [63].

GPFS makes use of a much smaller stripe size than Lustre (typically 16 KB or 64 KB) and sets the stripe width adaptively. For large parallel writes, data can be striped across all available GPFS servers, potentially providing a much greater maximum bandwidth [63].

3.3.3 The Parallel Log-structured File System

On top of parallel file systems like Lustre or GPFS, virtual file systems may provide an additional performance boost by transforming parallel file operations to be more appropriate for the underlying file system. One such example of this is the parallel log-structured file system (PLFS) [11] developed at the Los Alamos National Laboratory (LANL).

PLFS is a virtual file system that makes use of file partitioning and a log-structure (as described in Section 2.2.2) to improve the performance of parallel file operations. Each file within the PLFS mount point appears to an application as though it is a single file; PLFS, however, creates a container structure, with

	Minerva	Sierra	Cab
Processor	Intel Xeon 5650	Intel Xeon 5660	Intel Xeon E5-2670
CPU Speed	2.66 GHz	2.8 GHz	2.6 GHz
Cores per Node	12	12	16
Memory per Node	24 GB	24 GB	32 GB
Nodes	492	1,856	1,200
Interconnect	— QLogic TrueScale	4× QDR InfiniBand	—
File System	See Table 3.2	See Table 3.3	See Table 3.3

Table 3.1: Hardware specification of the Minerva, Sierra and Cab supercomputers.

Minerva File System		
File System		GPFS
I/O servers		2
Theoretical Bandwidth ^a		≈4 GB/s
	Storage	Metadata
Number of Disks	96	24
Disk Size	2 TB	300 GB
Spindle Speed	7,200 RPM	15,000 RPM
Bus Connection	Nearline SAS	SAS
RAID Configuration	Level 6 (8 + 2)	Level 1+0

Table 3.2: Configuration for the GPFS installation connected to Minerva.

^aTheoretical Bandwidth refers the maximum rate at which data can be transferred to the file servers and is therefore bounded only by the network interconnect.

a data file and an index for each process or compute node. This provides each process with its own unique file stream, potentially increasing the available bandwidth. Figure 3.9 demonstrates how a six rank (two processes per rank) execution would view a single file and how it would be stored within the PLFS backend directory.

In order to use PLFS on a supercomputer, either: the FUSE file system driver must be installed; a custom MPI library must be built; or applications must be rewritten to use the PLFS API directly. In Chapter 5 an alternative solution is provided, in addition to an in-depth investigation into why PLFS achieves the performance gains reported by its developers [11].

3.4 Computing Platforms

The work presented in this thesis has been carried out on four distinct HPC systems. Three of these are built from commodity hardware, one is a machine installed at the University of Warwick and the other two systems are installed

OCF <i>lscratchc</i> File System				
2011–2012			2013	
File System	Lustre		Lustre	
I/O Servers	24		32	
Theoretical Bandwidth ^a	≈ 30 GB/s		≈ 30 GB/s	
	Storage	Metadata	Storage	Metadata
Number of Disks	3,600	30 (+2) ^b	4,800	30 (+2) ^b
Disk Size	450 GB	147 GB	450 GB	147 GB
Spindle Speed	10,000 RPM	15,000 RPM	10,000 RPM	15,000 RPM
Bus Connection	SAS	SAS	SAS	SAS
RAID Configuration	Level 6 (8 + 2)	Level 1+0	Level 6 (8 + 2)	Level 1+0

Table 3.3: Configuration for the *lscratchc* Lustre File System installed at LLNL in 2011 (for the experiments in Chapter 5) and 2013 (for the experiments in Chapter 6).

^aTheoretical Bandwidth refers the maximum rate at which data can be transferred to the file servers and is therefore bounded only by the network interconnect.

^bThe MDS used by OCF’s *lscratchc* file system uses 32 disks: two configured in RAID-1 for journalling data, 28 disks configured in RAID-1+0 for the data volume itself and a further two disks to be used as hot spares.

at the Lawrence Livermore National Laboratory (LLNL) in the United States. The final machine used was the now decommissioned IBM BlueGene/P (BG/P) system that was installed at the Daresbury Laboratory in the United Kingdom. Specifically, the machines are:

Minerva

A capacity (used for many small tasks) supercomputer installed at the Centre for Scientific Computing within the University of Warwick. Minerva is an IBM iDataPlex system consisting of 492 nodes, each containing two hex-core Westmere-EP processors clocked at 2.66 GHz. The system is served by a small GPFS installation and the nodes are connected via QLogic’s TrueScale 4× QDR InfiniBand. The full specification can be found in Tables 3.1 and 3.2.

Sierra

A capability (used for a few very large tasks) HPC system installed in the Open Compute Facility (OCF) at LLNL. Sierra is a Dell Xanadu 3 Cluster consisting of 1,856 compute nodes, each containing two hex-core Westmere-EP processors running at 2.8 GHz. The interconnect is a QLogic QDR InfiniBand fat-tree (very similar to Minerva). Sierra is

connected to LLNL’s “islanded I/O” network, and can therefore make use of various different Lustre installations. In this thesis, work has been predominantly performed on the *lscratchc* file system due to its locality to Sierra. The experiments on Sierra were all performed prior to 2013, when the *lscratchc* file system was upgraded from 360 to 480 OSTs. More details can be found in Tables 3.1 and 3.3.

Cab

A capacity supercomputer installed in the OCF at LLNL. Cab is a Cray-built Xtreme-X cluster with 1,200 batch nodes, each containing two oct-core Xeon E5-2670 processors clocked at 2.6 GHz. An InfiniBand fat-tree connects each of the nodes and, like Sierra, Cab is connected to LLNL’s islanded I/O network. The work in this thesis was performed on the *lscratchc* file system after its upgrade to 480 OSTs. More information can be found in Tables 3.1 and 3.3.

BG/P

Daresbury’s BG/P system was a single cabinet, consisting of 1,024 compute nodes. Each node contained a single quad-processor compute card clocked at 850 MHz. The BlueGene/P architecture featured dedicated networks for point-to-point communications and MPI collective operations. File system and complex operating system calls (such as timing routines) were routed over the MPI collective tree to specialised higher-performance login or I/O nodes, enabling the design of the BlueGene compute node kernel to be significantly simplified to reduce background compute noise. The BG/P at Daresbury used a compute-node to I/O server ratio of 1:32; however, differing ratios were provided by IBM to support varying levels of workload I/O intensity. The BlueGene used in this thesis was supported by a GPFS storage solution with a hierarchical storage structure, where data was written to Fibre Channel disks initially (Stage 1 in Figure 3.5) before being staged onto slower SATA connected hard disks later (Stage 2

STFC BlueGene Platform	
Processor	PowerPC 450
CPU Speed	850 MHz
Cores per Node	4
Nodes	1,024
Interconnects	3D Torus
	Collective Tree
Storage System	See Table 3.5

Table 3.4: Hardware configuration for the IBM BlueGene/P system at the Daresbury Laboratory.

STFC BlueGene Platform File System		
File System	GPFS	
I/O servers	4	
Theoretical Bandwidth ^a	≈ 6 GB/s	
	Stage 1	Stage 2
Number of Disks	110	35
Disk Size	147 GB	500 GB
Spindle Speed	15,000 RPM	7,200 RPM
Bus Connection	Fibre Channel	SATA
RAID Configuration	Level 6 (8 + 2)	Level 5 (4 + 1)

^aTheoretical Bandwidth refers the maximum rate at which data can be transferred to the file servers and is therefore bounded only by the network interconnect.

Table 3.5: Configuration for the GPFS installation connected Daresbury Laboratory’s BlueGene/P, where data is first written to Fibre Channel connected disks because being staged to slower SATA disks.

in Figure 3.5). Furthermore, data and metadata were stored on the same storage medium. Daresbury’s BG/P compute and I/O configuration is summarised in Tables 3.4 and 3.5, respectively.

3.5 Input/Output Benchmarking Applications

Throughout this thesis, work has been performed using a variety of different benchmarks. Specifically, this thesis makes extensive use of four benchmarks which are representative of a broad range of high performance applications. These applications are:

IOR

A parameterised benchmark that performs I/O operations through both the HDF-5 and the MPI-IO interfaces [120, 121]. The application can be configured to be representative of a large number of science applications

with minimal configuration.

FLASH-IO

A benchmark that replicates the HDF-5 checkpointing routines found in the FLASH [4, 155] thermonuclear star modelling code [47, 109]. The local problem size can be configured at compile time to behave in the same way as any given FLASH dataset.

BT

An application from the NAS Parallel Benchmark (NPB) Suite which has been configured by NASA to replicate I/O behaviour from several important internal production codes [7, 8].

mpi_io_test

A parameterised benchmark developed at LANL, primarily used for benchmarking the performance of PLFS. In particular, `mpi_io_test` provides an interface for writing *N-to-N*, *N-to-M* and *N-to-1*, allowing for a comparison of writing techniques [95].

Of these four applications, two are standard benchmarks used for the assessment of parallel file systems (IOR and `mpi_io_test`), while the other two have been chosen as they recreate the I/O behaviour of much larger codes but with a reduced compute time and less configuration than their parent codes (FLASH-IO and BT). This permits the investigation of system configurations that may have an impact on the I/O performance of the larger codes, without requiring considerable machine resources.

In addition to these applications, a custom benchmark has been written to assess the impact of some of the tools presented in this thesis on the performance of the MPI communication library (see Chapter 4). A further benchmarking application has also been written to explore the effect of contention on the Lustre file system (see Chapter 6).

3.6 Summary

The hardware and software in use on modern supercomputers varies drastically between different organisations and installations, but the principles that dictate performance remain largely the same. In this chapter the history and structure of I/O in parallel computation has been described, starting with the development and improvement of HDDs (which continue to dominate HPC I/O installations [151]), to the creation of the first networked file system, and up to the DFSs in use at the time of writing.

Modern parallel file systems make use of an object-based storage approach which is not dissimilar to the operation of standalone file systems such as ext4. Files are divided into discrete blocks and, where on standalone file systems these blocks are spread across a single disk, on a DFS the blocks are distributed amongst several separate disks and file servers. The structure of these files and the properties associated with them are then stored in a metadata database, which may itself be distributed.

With the decreasing cost of solid state drives, their use in HPC installations is increasing. Modern HPC systems are beginning to combine both HDDs and SSDs into tiered architectures – using SSDs as a staging area, before committing data to slower HDDs at a non-critical time [151]. The idea of using tiered storage is not new and is used in the BlueGene/P used in this thesis – where data is written initially to fast Fibre Channel disks, before being moved to slower disks. The use of tiered/hybrid I/O systems is changing the performance characteristics of parallel file systems [138]. However, much of the work in this thesis will similarly apply when SSD adoption increases; ensuring data consistency through the use of file locking will still reduce the performance of large distributed writes and contention will still hamper the performance of shared file systems, albeit to a lesser extent.

The primary purpose of modern day parallel storage for science applications is to provide an interface through which applications can store the results of

long-running computations. The data generated by these applications can be used for additional purposes beyond producing the answers to important scientific questions. Data written throughout an execution of a scientific application can be used to visualise the progression of a computation and to facilitate software resilience. It is estimated that on exascale machines, applications may have to survive multiple node failures per day; the focus of this thesis is on the checkpointing routines that are used in scientific applications to provide snapshots, enabling application state recovery following a failure.

Throughout this thesis, multiple applications and hardware systems are used to assess the current state of parallel I/O and how it must adapt to solve the challenges exascale computation will bring. The hardware configurations used throughout the remainder of this thesis are summarised in this chapter, along with the applications that are used to assess them.

CHAPTER 4

I/O Tracing and Application Optimisation

As the HPC industry moves towards exascale computing, the increasing number of compute components will have huge implications for system reliability. As a result, checkpointing – where the system state is periodically written to persistent storage so that, in the case of a hardware or software fault, the computation can be restored and resumed – is becoming common-place. The cost of checkpointing is a slowdown at specific points in the application in order to achieve some level of resilience. Understanding the cost of checkpointing, and the opportunities that might exist for optimising this behaviour, presents a genuine opportunity to improve the performance of parallel applications at scale.

Performing I/O operations in parallel using MPI-IO or file format libraries, such as the hierarchical data format (HDF-5), has partially encouraged code designers to treat these libraries as a black box, instead of investigating and optimising the data storage operations required by their applications. Their focus has largely been improving compute performance, often leaving data-intensive operations to third-party libraries. Without configuring these libraries for specific systems, the result has often been poor I/O performance that has not realised the full potential of expensive parallel disk systems [9, 10, 66, 141, 146].

This chapter documents the design, implementation and application of the RIOT I/O Toolkit (referred to throughout the remainder of this thesis by the recursive acronym RIOT), described previously [141, 142, 144] to demonstrate the I/O behaviours of three standard benchmarks at scale on three contrasting HPC systems. RIOT is a collection of tools developed specifically to enable the tracing and subsequent analysis of application I/O activity. The tool is able to

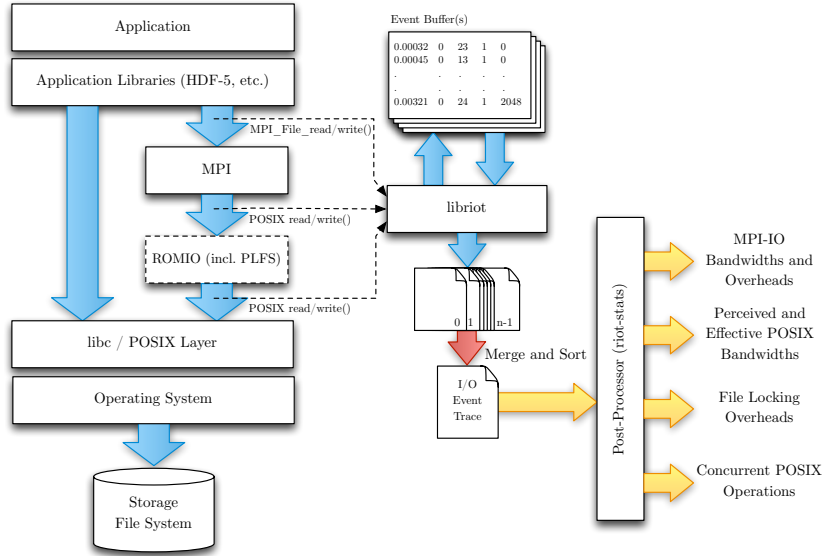


Figure 4.1: Tracing and analysis workflow using the RIOT toolkit.

trace parallel file operations performed by the ROMIO layer (see Section 2.2.3 for details) and relate these to their underlying POSIX file operations. This recording of low-level parameters permits analysis of I/O middleware, file format libraries, application behaviour and to some extent even the underlying file systems used by large clusters.

4.1 The RIOT I/O Toolkit

The left-hand side of Figure 4.1 depicts the usual flow of I/O in parallel applications; generally, applications either use the MPI-IO file interface directly, or use a third-party library such as HDF-5 or NetCDF. In both cases, MPI is ultimately used to perform the read and write operations. In turn, MPI calls upon the MPI-IO library which, in the case of both OpenMPI and MPICH, is the ROMIO implementation [127]. The ROMIO file system driver [125] then calls the file system’s operations to read/write the data from/to the file system.

RIOT is an I/O tracing tool that can be used either as a dynamically loaded library (via runtime pre-loading and linking) or as a static library (linked at

compile time). In the case of the former, the shared library uses function interpositioning to place itself in to the library stack immediately prior to execution. When compiled as a dynamic library, RIOT redefines several functions from the POSIX API and MPI libraries – when the running application makes calls to these functions, control is instead passed to handlers in the RIOT library. These handlers allow the original function to be performed, timed and recorded into a log file for each MPI rank. By using the dynamically loadable `libriot`, application recompilation is avoided completely; RIOT is therefore able to operate on existing application binaries and remain agnostic to compiler and implementation language.

For situations where dynamic linking is either not desirable or is only available in a limited capacity (such as in the BG/P system used in this study), a static library can be built. The RIOT software makes use of macro functions in order to control how the library is built (i.e. whether a statically linked library or a dynamically loadable library should be built). A compiler wrapper is then used to compile RIOT into a parallel application using the `-wrap` functionality found in the Linux linker. Listing 4.1 shows how one function (namely the `MPI_File_open()` function) looks within RIOT.

As shown in Figure 4.1, `libriot` intercepts I/O calls at three positions. In the first instance, MPI-IO calls are intercepted and redirected through RIOT, using either the PMPI interface, or via dynamic or static linking; in the second instance, POSIX calls made by the MPI library are intercepted; and in the final instance, any POSIX calls made by the ROMIO file system interface are caught and processed by RIOT.

Traced events in RIOT are recorded in a buffer stored in main memory. While the size of the buffer is configurable, experiments have suggested that a buffer of 8 MB is sufficient for the experiments in this thesis and adds minimal overhead to the application. A buffer of this size allows approximately 340,000 file operations to be stored before needing to be flushed to the disk. This delay of logging (by storing events in memory) may have a small effect on compute

```
int FUNCTION_DECLARE(MPI_File_open)(MPI_Comm comm, char *filename,
                                   int amode, MPI_Info info, MPI_File *fh) {
    // The FUNCTION_DECLARE macro controls how
    // functions are defined, depending on if the static or
    // dynamic library is being built.
    DEBUG_ENTER;

    // Maps the real MPI_File_open command to __real_MPI_File_open
    MAP(MPI_File_open);

    // Add file to the database
    int fileid = addFile(filename);

    // Add a record to the log
    addRecord(BEGIN_MPI_OPEN, fileid, 0);

    // Perform correct operation
    int ret = __real_MPI_File_open(comm, filename,
                                   amode, info, fh);

    // Add a end record to the log
    addRecord(END_MPI_OPEN, fileid, 0);

    DEBUG_EXIT;
    return ret;
}
```

Listing 4.1: Source code demonstrating how the `MPI_File_open` function is interpositioned in RIOT.

performance (since the memory access patterns may change), but storing trace data in memory helps to prevent any distortion of application I/O performance. In the event that the buffer becomes full, the data is written out to disk and the buffer is reset. This repeats until the application has terminated.

Time consistency is established across multiple nodes by overloading the `MPI_Init()` function to force all ranks to wait at the start of execution on an `MPI_Barrier()` before each resetting their respective timers; after this initial barrier, each rank can progress uninterrupted by RIOT. This is especially important on architectures such as IBM's BlueGene, as applications can take several minutes to start across the whole cluster. Synchronising in this manner enables more accurate ordering of events even if nodes have experienced a significant degree of time drift.

After the recording of an application trace is complete, a post-execution analysis phase can be conducted (see Figure 4.1).

During execution, RIOT builds a file lookup table and for each operation only stores the time, the rank, a file identifier, an operation identifier and the file offset. After execution, these log files are merged and time-sorted into a single master log file, as well as a master file database.

Using the information stored, RIOT can:

- Produce a complete runtime trace of an application’s I/O behaviour;
- Demonstrate the file locking behaviour of a particular file system;
- Calculate the effective POSIX bandwidth achieved by MPI to the file system;
- Visualise the decomposition of an MPI file operation into a series of POSIX operations; and,
- Demonstrate how POSIX operations are queued and then serialised by the I/O servers.

Throughout this thesis, a distinction is made between effective MPI-IO and POSIX bandwidths – *MPI-IO bandwidths* refer to the data throughput of the MPI functions on a per MPI-rank basis. *POSIX bandwidths* relate to the data throughput of the POSIX read/write operations as if performed serially and called directly by the MPI library. This distinction is made due to the inability to accurately report the perceived POSIX bandwidth because of the non-deterministic nature of parallel POSIX writes. The perceived POSIX bandwidth is therefore bounded below by the *perceived* MPI bandwidth (since the POSIX bandwidths must necessarily be at least as fast as the MPI bandwidths), and is bounded above by the *effective* POSIX bandwidth multiplied by the number of ranks (assuming a perfect parallel execution of each POSIX operation).

4.1.1 Feasibility Study

To ensure RIOT does not significantly affect the runtime behaviour and performance of scientific codes, an I/O benchmark has been specifically designed to

assess the overheads introduced by the use of RIOT. The application performs a known set of read and write operations over a series of files. Each process performs 100 read and write operations in 4 MB blocks. The benchmark application was executed on three of the test platforms used in this thesis in three distinct configurations: (i) without RIOT; (ii) with RIOT configured to only trace POSIX file operations; and, (iii) with RIOT performing a complete trace of MPI and POSIX file activity. The six MPI operations chosen for this feasibility study were: `MPI_File_read/write()`, `MPI_File_read_all/write_all()` and `MPI_File_read_at_all/write_at_all()`; analysis of the scientific codes used throughout this thesis, and other similar applications, suggests that these functions are amongst the most commonly used for performing parallel I/O (see Appendix A for more details).

Figure 4.2 shows the time taken to perform 100 `MPI_File_write_all()`, and `MPI_File_read_all()` operations at differing core counts (results for additional functions are shown in Appendix A). From these experiments it is clear that RIOT adds minimal overhead to an application's runtime, although it is particularly difficult to precisely quantify this overhead since the machines employed operate production workloads.

As shown by the confidence intervals in Figure 4.2, on Minerva, repeated runs produce nearly identical results due to the relatively small size of the machine and the lack of heavy utilisation on the I/O backplane. For Sierra, results vary more widely due to several I/O intensive applications running on the same storage subsystem simultaneously. On BG/P the results are similarly varied, and in some cases the application runs vary more widely due to the use of I/O aggregator nodes in addition to the compute nodes. Nevertheless, the results of these experiments show that the average overhead of RIOT is rarely greater than 5% for `MPI_File` operations. Low overhead tracing is a key feature in the design of RIOT, and is an important consideration for profiling activities associated with large codes that may already take considerable lengths of time to run in their own right.

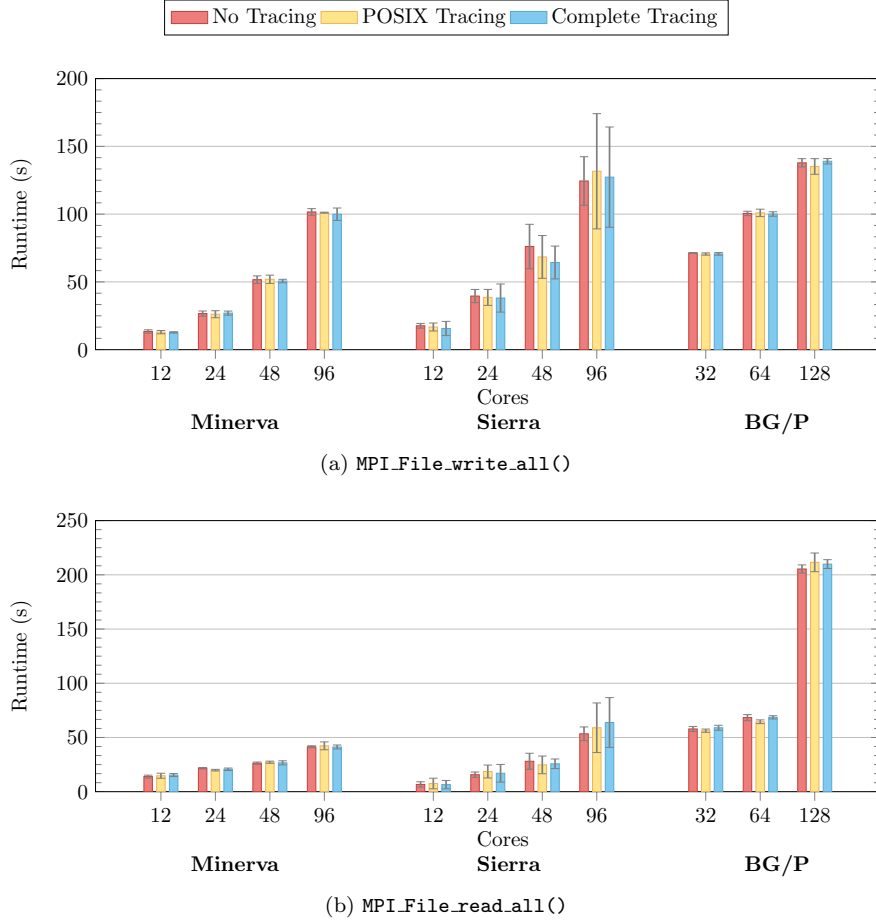


Figure 4.2: Total runtime of RIOT overhead analysis benchmark for the functions `MPI_File_write_all()` and `MPI_File_read_all()`, on three platforms at varying core counts, with three different configurations: No RIOT tracing, POSIX RIOT tracing and complete RIOT tracing.

4.2 File System Analysis

One key use-case of RIOT is to trace the write behaviour of scientific codes. To demonstrate this, analysis has been performed on three distinct codes (one of which was executed in two different configurations). Each of the codes were executed using the default configuration options for the test machine in question. For both Minerva and Sierra, data was pushed to the disks using the UNIX File System (UFS) MPI-IO driver (`ad_ufs`). For Minerva, data was striped across its two servers with metadata operations being distributed between these

two servers. For Sierra, metadata operations were performed on a dedicated metadata server, while data was by striped across two OSTs.

4.2.1 Distributed File Systems – Lustre and GPFS

As outlined in Chapter 3, the three test clusters employed in this chapter make use of two different file systems – both Minerva and BG/P make use of GPFS, while Sierra uses a Lustre installation. The I/O backplane used by Minerva and that used by Sierra may seem vastly different, but the default configuration of *lscratchc* means that the performance of both are similar since in each case, files are striped usually over two OSTs. Both GPFS installations adapt to their workload, though as stated previously, this usually means striping data over the two available servers in Minerva’s case. As demonstrated in Figure 4.3(a), at low core counts Sierra achieves the fastest write speed for IOR using MPI-IO, though this is soon exceeded by BG/P as the number of cores is increased. Figure 4.3 shows that for IOR and FLASH-IO, Minerva’s performance follows the trend of Sierra, though performs slightly worse due to the slower hardware being employed.

It is interesting to note that IOR writing through the HDF-5 middleware library (Figure 4.3(b)) exhibits very different performance to the same benchmark running with only MPI-IO, despite writing similar amounts of data to the same offsets on both Sierra and Minerva. The performance of FLASH-IO (Figure 4.3(c)) also suggests that a significant performance defect exists in the HDF-5 library. On each of these systems, the parallel HDF-5 library, by default, attempts to use data-sieving in order to transform many discontinuous small writes into a single much larger write. In order to do this, a large region (containing the target file locations) is locked and read into memory. The small changes are then made to the block in memory, and the data is then written back out to persistent storage in a single write operation. While this offers a large improvement in performance for small unaligned writes [126], many HPC applications are constructed to perform larger sequential file operations.

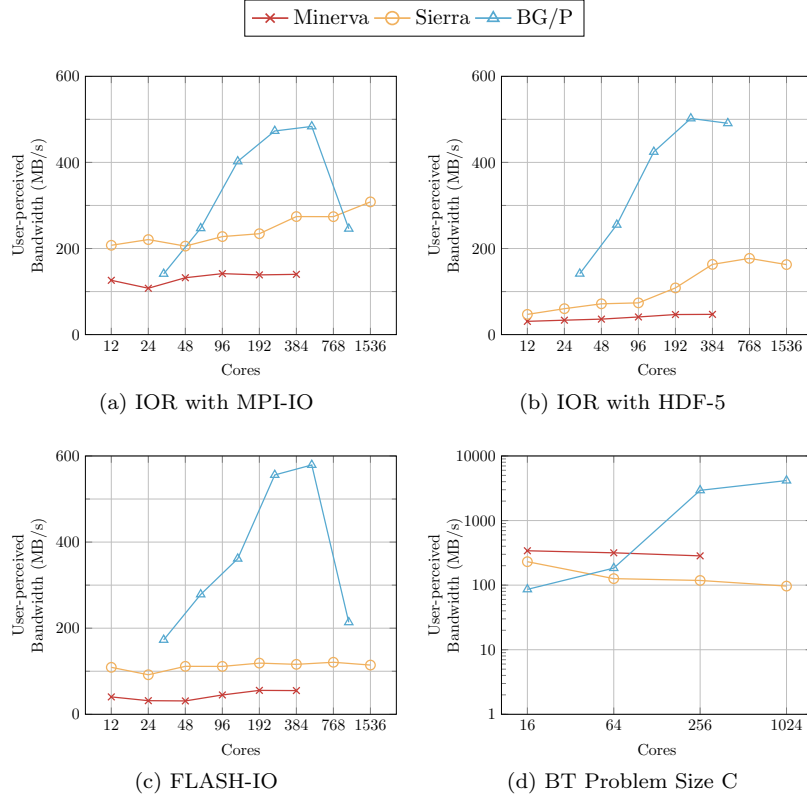


Figure 4.3: User-perceived bandwidth for applications on the three test systems.

When using data-sieving, the use of file locks helps to maintain file coherence. However, as RIOT is able to demonstrate, when writes do not overlap, the locking, reading and unlocking of file regions may create a significant overhead – this is discussed further in Section 4.3.

The results in Figure 4.3(d) show that the BT mini-application achieves by far the greatest performance on all three test systems (note the logarithmic scale). On the BG/P system, its performance at 256 cores is significantly greater than at 64 cores. Due to the architecture of the machine and the relatively small amount of data that each process writes at this scale, the data is flushed very quickly to the I/O node’s cache and this gives the illusion that the data has been written to disk at speeds in excess of 1 GB/s. For much larger output sizes the same effect is not seen, since the writes are much larger and therefore cannot be flushed to the cache at the same speed. This is demonstrated in the

performance of IOR (Figures 4.3(a) and 4.3(b)) and FLASH-IO (Figure 4.3(c)).

Note that while the I/O performance of Minerva and Sierra plateau quite early, the I/O performance of the BG/P system does not. A commodity cluster using MPI will often use ROMIO hints such as *collective buffering* [92] to reduce the contention for the file system; the BG/P performs what could be considered “super” collective buffering, where 32 nodes send all of their I/O traffic through a single aggregator node. In addition to this, BG/P also uses faster disks and a purpose written MPI-IO file system driver (`ad_bg1`). The exceptional scaling behaviour observed in Figure 4.3(d) can be attributed to this configuration. As the output size and the number of participating nodes increases, contention begins to affect performance.

Although the configuration of the BlueGene’s file system was somewhere between that of Sierra and Minerva, it provided twice the number of file servers as Minerva and therefore striped its data over four servers instead of two. Additionally, the disks were configured such that data was committed first to Fibre Channel connected hard disk drives, before being staged to slower SATA disks. The use of a tiered file system (where the I/O is performed from dedicated nodes to FC-connected burst buffers, before being committed to SATA disks) and MPI-IO features such as collective buffering and data-sieving (which can be done at an I/O node level, rather than on each compute node) enabled the BG/P’s GPFS installation to perform far better than the other file systems.

The write performance on each of the commodity clusters is roughly $2 - 3\times$ the write speed of a single consumer-grade hard disk. Considering that these systems consist of hundreds (or thousands) of disks, configured to read and write in parallel, it is clear that the full potential of the hardware is not being realised with the current configurations. Analysing the effective bandwidth of each of the codes (i.e. the total amount of data written, divided by the total time taken by all nodes) shows that data is being written very slowly to the individual disks when running at scale. The effective MPI and POSIX bandwidth achieved by each of the applications can be seen in Figures 4.4, 4.5,

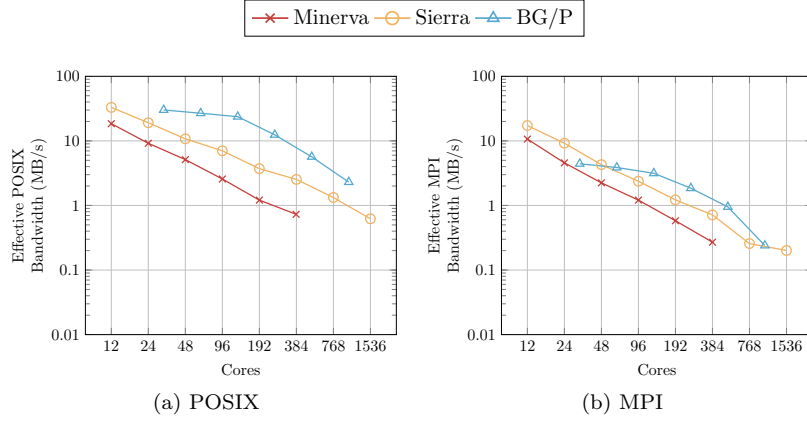


Figure 4.4: Effective POSIX and MPI bandwidth for IOR through MPI-IO.

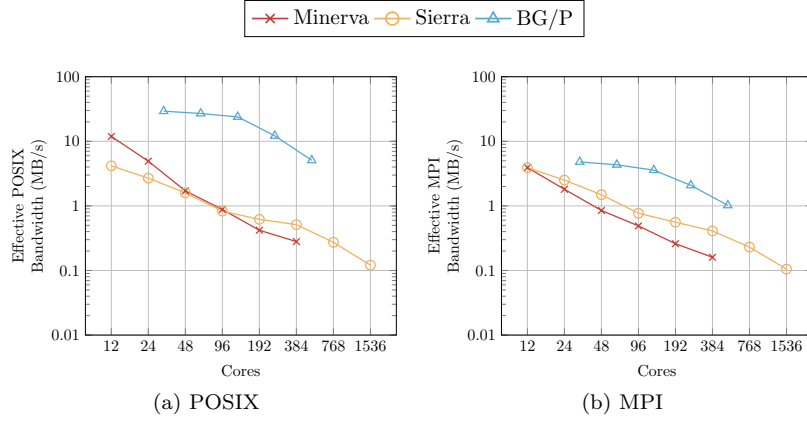


Figure 4.5: Effective POSIX and MPI bandwidth for IOR through HDF-5.

4.6 and 4.7. While one would expect the POSIX bandwidth to slightly exceed the MPI bandwidth (due to a small processing overhead in the MPI library), the degree to which this is true demonstrates a much larger than expected overhead in the MPI library. For IOR, using MPI-IO directly (Figure 4.4), on Minerva, the effective POSIX bandwidth is often more than twice the effective MPI bandwidth, but peaks at only 11.105 MB/s for the single node case. For the much larger Sierra supercomputer, for the single node case the effective MPI and POSIX bandwidths are almost equivalent but again peak at only 4.173 MB/s. Figures 4.5, 4.6 and 4.7 demonstrate a similar trend, showing that the low effective POSIX bandwidth achieved does not nearly approach the potential performance of each storage system.

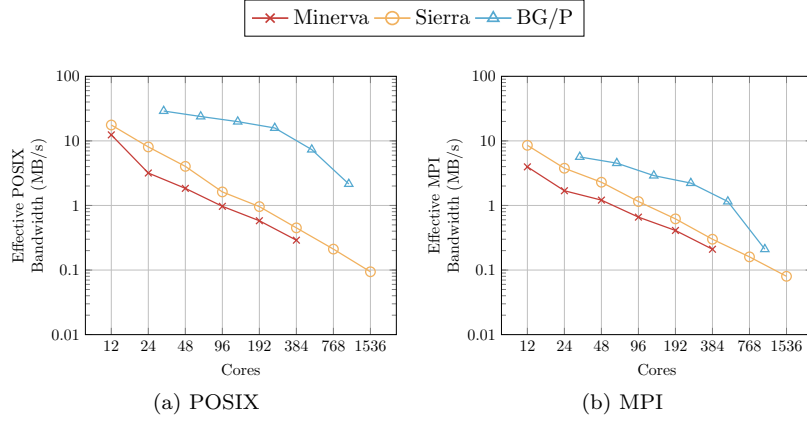


Figure 4.6: Effective POSIX and MPI bandwidth for FLASH-IO.

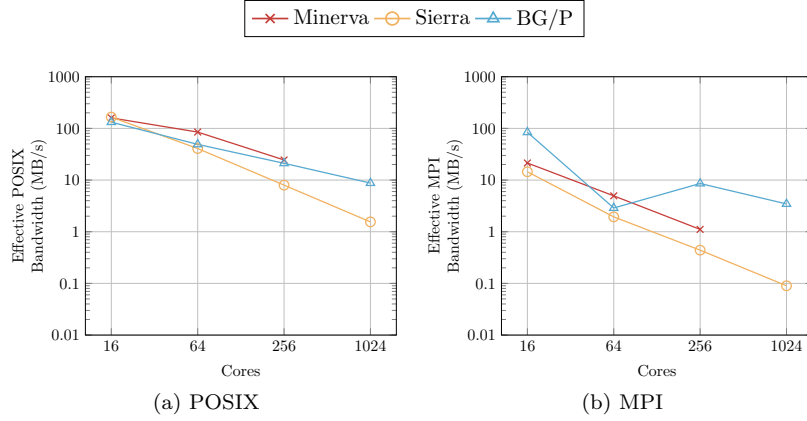


Figure 4.7: Effective POSIX and MPI bandwidth for BT Problem C, as measured by RIOT.

On the Lustre system data is striped across two OSTs, where each OST is a RAID-6 caddy consisting of 10 disk drives. As the disks are Serial Attached SCSI (SAS), each individual disk should have a maximum bandwidth of either 150 MB/s or 300 MB/s, giving a maximum potential bandwidth of 1,200 MB/s or 2,400 MB/s¹. While increasing the amount of parallelism in use for computation reduces the time to solution for applications, as the storage resource in use are not similarly scaled, the added contention harms the storage performance. On the GPFS systems, similar effective bandwidth is shown, though the number of storage targets data is striped across is not known, as GPFS stripes

¹The SAS version in use on *lscratch* is unknown, and therefore may run at 3.0 Gbit/s or 6.0 Gbit/s.

dynamically. This poor level of performance may be partially attributed to two problems: (i) disk seek time, and (ii) file system contention. In the former case, since data is being accessed simultaneously from many different nodes and users, the file servers must constantly seek for the information that is required. In the latter case, since reads and writes to a single file must maintain some degree of consistency, contention for a single file can become prohibitive.

From the results presented in Figure 4.3 and Appendix B, it is clear that Sierra generally has a much higher performance I/O subsystem than Minerva. However, the BG/P's file system far outperforms both clusters when scaled. The unusual interconnect and architecture that it uses allows its compute nodes to flush their data to the I/O aggregator's cache quickly, allowing computation to continue. Similarly, when the writes are small, Minerva can be shown to outperform Sierra, mainly due to the locality of its I/O backplane. However, when HDF-5 is in use on Minerva, the achievable bandwidth is much lower than that of the other machines due to file-locking and the poor read performance of its hard disk drives.

Ultimately, both Sierra and Minerva exhibit similar performance (as expected by using only two OSTs of *lscratchc*). However, Sierra's performance does slightly exceed Minerva's in almost all cases due to the use of faster enterprise-class disks and centralised metadata storage, decreasing the amount of processing each that OSS has to perform. The BG/P solution exhibits the greatest performance due to the use of four OSSs, fibre channel connected disks, and dedicated I/O aggregator nodes. As demonstrated in the next section, when the I/O operations required are analysed and well understood, better performance can be achieved on both Lustre and GPFS with minimal effort.

4.3 Middleware Analysis and Optimisation

The experiments with FLASH-IO and IOR, both through HDF-5, demonstrate that a large performance gap exists between using the HDF-5 file format li-

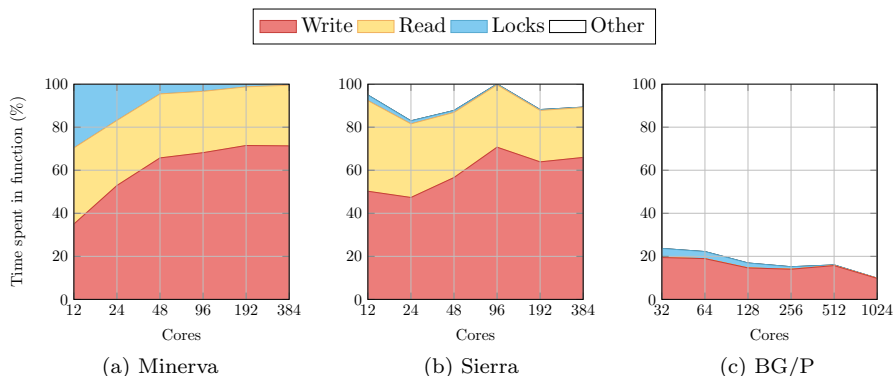


Figure 4.8: Percentage of time spent in POSIX functions for FLASH-IO on three platforms.

brary and performing I/O directly via MPI-IO. While a slight slowdown may be expected, since there is an additional layer of abstraction in the software stack to traverse, the decrease in performance is quite large (up to a 50% slowdown). Figure 4.8 shows the percentage of time spent in each of the four main contributing POSIX functions to `MPI_File_write` operations.

For the Minerva supercomputer, at low core counts, there is a significant overhead associated with file locking (Figure 4.8(a)). In the worst case, on a single node, this represents an approximate 30% decrease in performance. The reason for the use of file locking in HDF-5 is that data-sieving is used by default to write small unaligned blocks in much larger blocks. The penalty for this is that data must be read into memory prior to writing; this behaviour can prove to be a large overhead for many applications, where the writes may perform much better were data-sieving to be disabled. Figure 4.8(c) shows that the BG/P does not perform data-sieving, as evidenced by the lack of read functions. However, due to the use of dedicated I/O nodes, the compute nodes spend approximately 80% of their MPI write time waiting for the I/O nodes to complete.

In contrast to Minerva, the same locking overhead is not experienced by Sierra; however up to 20% of the MPI write time is spent waiting for other ranks. It is also of note that Minerva’s storage subsystem is backed by relatively slow HDDs; Sierra on the other hand uses much quicker enterprise-class drives,

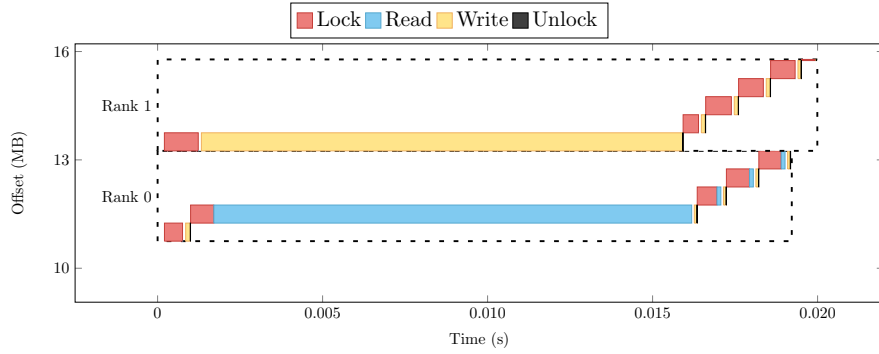


Figure 4.9: Composition of a single, collective MPI write operation on MPI ranks 0 and 1 of a two core run of FLASH-IO, called from the HDF-5 middleware library in its default configuration.

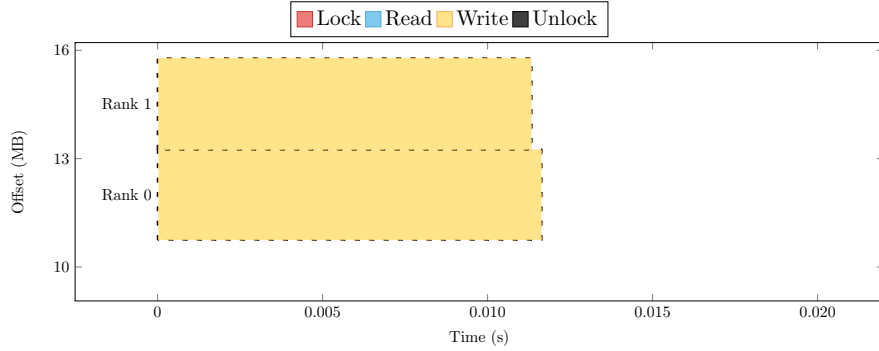


Figure 4.10: Composition of a single, collective MPI write operation on MPI ranks 0 and 1 of a two core run of FLASH-IO, called from the HDF-5 middleware library after data-sieving has been disabled.

providing a much smaller seek time, a much greater bandwidth and various other performance advantages (e.g. greater rotational vibration tolerance, larger cache, etc.). As a consequence of this, a single Sierra I/O node can service a read request much more quickly than one of Minerva's, providing an overall greater level of service.

Using RIOT's tracing and visualisation capabilities, the execution of a small run of the FLASH-IO benchmark (using a $16 \times 16 \times 16$ grid size and only two cores) can be investigated. Figure 4.9 shows the composition of a single MPI-IO write operation in terms of its POSIX operations. Rank 0 spends the majority of its `MPI_File_write` time performing read, lock and unlock operations,

whereas Rank 1 spends much of its time performing only lock, unlock and write operations. Since Rank 1 writes to the end of the file, increasing the end-of-file pointer, there is no data for it to read in during data-sieving; Rank 0, on the other hand, will always have data to read, as Rank 1 will have increased the file size, effectively creating zeroed data between Rank 0's position and the new end-of-file pointer.

Both ranks splitting one large write into five “lock, read, write, unlock” cycles is indicative of using data-sieving, with the default 512 KB buffer, to write approximately 2.5 MB of data. When performing a write of this size, where all the data is “new”, data-sieving may be detrimental to performance. In order to test this hypothesis the `MPI_Info_set` operations present in the FLASH-IO source code (used to set the MPI-IO hints) can be modified to disable data-sieving. Figure 4.10 shows that, with the modified configuration, the MPI-IO write operation is consumed by a single write operation, and the time taken to perform the write is 40% shorter than that found in Figure 4.9.

Using the problem size benchmarked in Figures 4.3 and 4.6 ($24 \times 24 \times 24$), the original experiments were repeated on both Minerva and Sierra using between 1 and 32 compute nodes (12 to 384 cores) in three configurations: firstly, in the original configuration; secondly, with data-sieving disabled; and, finally, with collective buffering enabled and data-sieving disabled. Figure 4.11(a) demonstrates the resulting improvement on Minerva, showing a $2\times$ increase in write bandwidth over the unmodified code. Better performance is observed when using collective buffering. On Sierra (Figure 4.11(b)) there is a similar improvement in performance (approximately $2\times$ increase in bandwidth). On a single node (12 cores), performing only data-sieving is slightly faster than using collective buffering, and beyond this collective buffering increases the bandwidth by between 5% and 20% (numeric data and confidence intervals are shown in Appendix C). Of particular note is the performance at 384 cores, where disabling collective buffering increases performance; however, the increased variance in the results at this scale indicates that this may be a side effect of background

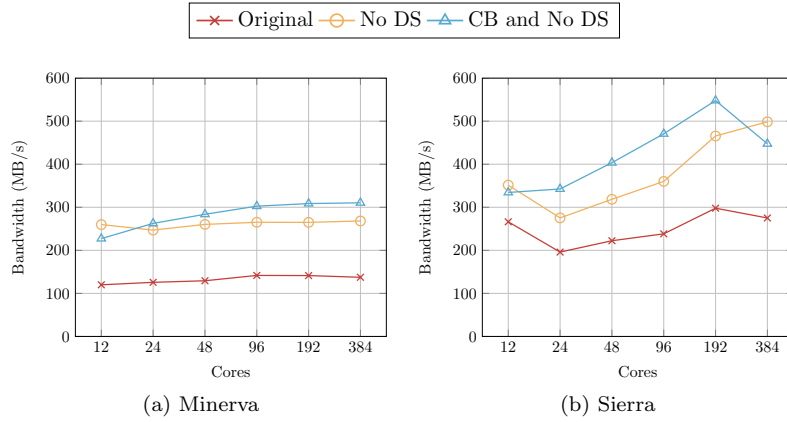


Figure 4.11: Perceived bandwidth for the FLASH-IO benchmark in its original configuration (Original), with data-sieving disabled (No DS), and with collective buffering enabled *and* data-sieving disabled (CB and No DS) on Minerva and Sierra, as measured by RIOT.

machine noise.

This result does not mean that data-sieving will always decrease performance; in the case that data in an output file is being updated (rather than a new output file generated), using data-sieving to make small differential changes may improve performance [26].

4.4 Summary

Parallel I/O operations continue to represent a significant bottleneck in large-scale parallel scientific applications. This is, in part, because of the slower rate of development that parallel storage has witnessed when compared to that of microprocessors. Other causes include limited optimisation at code level and the use of complex file formatting libraries. Contemporary applications can often exhibit poor I/O performance because code developers lack an understanding of how their code use I/O resources and how best to optimise for this.

In this chapter the design, implementation and application of RIOT has been presented. RIOT is a toolkit with which some of these issues might be addressed. RIOT's ability to intercept, record and analyse information relating to file reads, writes and locking operations has been demonstrated using three

standard industry I/O benchmarks. RIOT has been used on two commodity clusters as well as an IBM BG/P supercomputer.

The results generated by the tool illustrate the difference in performance between the relatively small storage subsystem installed on the Minerva cluster and the much larger Sierra I/O backplane. While there is a large difference in the size and complexity of these I/O systems, some of the performance differences originate from the contrasting hardware and file systems that they use and how the applications make use of these. Furthermore, through using the BG/P located at STFC Daresbury Laboratory, it has been shown that exceptional performance can be achieved on small I/O subsystems where dedicated I/O aggregators and tiered storage systems are used as burst buffers, allowing data to be quickly flushed from the compute node to an intermediate node.

RIOT provides the opportunity to:

- Calculate not only the bandwidth perceived by a user, but also the effective bandwidth achieved by the I/O servers. This has highlighted a significant overhead in MPI, showing that the POSIX write operations to the disk account for little over half of the MPI write time. It has also been shown that much of the time taken by MPI is consumed by file locking behaviours and the serialisation of file writes by the I/O servers.
- Demonstrate the significant overhead associated with using the HDF-5 library to store data grids. Through the data extracted by RIOT, it has been shown that on a small number of cores, the time spent acquiring and releasing file locks can consume nearly 30% of the file write time. Furthermore, on small-scale, multi-user I/O systems, reading data into memory before writing, in order to perform data-sieving, can prove very costly.

- Visualise the write behaviour of MPI when data-sieving is in use, showing how large file writes are segmented into many 512 KB lock, read, write, unlock cycles. Through adjusting the MPI hints to disable data-sieving it has been shown that on some platforms, and for some applications, data-sieving may negatively impact performance.

The investigation into the use of RIOT to analyse the behaviour of parallel storage continues in the next chapter, but already its use in identifying optimisation opportunities has been demonstrated. RIOT affords developers an opportunity to understand exactly how configuration options change the I/O behaviour and thus affect performance. By analysing the current performance behaviour of HDF-5 based applications a speed-up of at least $2\times$ can be achieved with a system's "stock" MPI installation, without affecting other applications or services on the system.

The results in this chapter have also highlighted the potential that exists in tiered storage systems, suggesting that these could very well be the answer to affordable, efficient and performant storage systems at exascale.

CHAPTER 5

Analysis and Rapid Deployment of the Parallel Log-Structured File System

As the performance of I/O systems continue to diverge substantially from that of the supercomputers that they support, a number of projects have been initiated to look for software- and hardware-based solutions to address this concern. One such solution is the parallel log-structured file system (PLFS) – which was created at the Los Alamos National Laboratory (LANL) [11] and is now being commercialised by EMC Corporation (EMC²). PLFS makes use of (i) a *log-structure*, where write operations are performed sequentially to the disk regardless of intended file offsets (keeping the offsets in an index structure instead) [108]; and (ii) *file partitioning*, where a write to a single file is instead transparently transposed into a write to many files, thus increasing the number of available file streams [135].

Currently PLFS can be deployed in one of three ways: (i) through a file system in userspace (FUSE) mount point, requiring installation and access to the FUSE Linux kernel module and its supporting drivers and libraries [42]; (ii) through an MPI-IO file system driver built into the Message Passing Interface (MPI) library [125]; or (iii) through the rewriting of an application to use the PLFS API directly [80]. These methods therefore require either the installation of additional software, recompilation of the MPI application stack (and, subsequently, the application itself) or modification of the application's source code. In HPC centres which have a focus on reliability, or which lack the time and/or expertise to manage the installation and maintenance of PLFS, it may be seen as too onerous to be of use.

In this chapter an analysis of PLFS is performed using RIOT in order to

demonstrate why PLFS increases the potential bandwidth available to applications. Due to the implications of installing and maintaining PLFS on a large system, an alternative approach to using PLFS is also presented [143]. This approach will facilitate rapid deployment of PLFS, and therefore allow application developers to accelerate their I/O operations without the burdens associated with PLFS installation. The techniques outlined are applicable to many virtual file systems and allow users to forgo the need to rewrite applications, obtain specific file/system access permissions, or modify the application stack.

5.1 Analysis of PLFS

The primary goal of PLFS is to intercept standard I/O operations and transparently translate them from N processes writing to a single file, to N processes writing to N files. The middleware creates a “view” over the N files, so that the calling application can operate on these files as if they were all concatenated into a single file. The use of multiple files by the PLFS layer helps to significantly improve file write times, as multiple, smaller files can be written simultaneously. Furthermore, improved read times have also been reported when using the same number of processes to read back the file as were used in its creation [103].

Table 5.1 presents the average perceived and effective MPI-IO and POSIX bandwidths achieved by the BT benchmark when running with the PLFS MPI-IO file system driver (`ad_plfs`) and without it, using the UNIX file system MPI-IO driver (`ad_ufs`). Note that, as previously, effective bandwidth in this table refers to the bandwidth of the operations as if called serially and hence are much lower than the perceived bandwidths.

As shown throughout Chapter 4, the effective POSIX write bandwidth decreases significantly as the size of application runs is increased. PLFS partially reverses this trend, as the individual POSIX writes are no longer dependent on operations performed by other processes (which are operating on their own files) and can therefore be flushed to the file server’s cache much more quickly. The

	16	ad_ufs 64	256	16	ad_plfs 64	256
Minerva						
User Perceived Bandwidth	252.888	233.456	173.696	397.579	440.546	660.373
<i>Speed-up</i>				1.572×	1.887×	3.802×
Effective POSIX Bandwidth	218.024	80.091	19.049	196.738	124.340	105.597
<i>Speed-up</i>				0.902×	1.552×	5.543×
Effective MPI Bandwidth	15.883	3.651	0.678	25.036	6.899	2.580
<i>Speed-up</i>				1.576×	1.900×	3.805×
Sierra						
User Perceived Bandwidth	212.486	126.102	115.191	405.495	1505.819	3122.271
<i>Speed-up</i>				1.908×	11.941×	27.105×
Effective POSIX Bandwidth	155.754	41.970	7.977	299.084	538.130	437.880
<i>Speed-up</i>				1.920×	12.822×	54.893×
Effective MPI Bandwidth	13.346	1.970	0.450	20.806	23.720	12.183
<i>Speed-up</i>				1.559×	12.041×	27.073×

Table 5.1: Perceived and Effective Bandwidth (MB/s) for BT class C through MPI-IO and PLFS, as well as the speed-up generated by PLFS.

log-structured nature of PLFS also increases the bandwidth, as data can be written in a non-deterministic sequential manner with a log file keeping track of the data ordering. For a class C execution on 256 cores, PLFS increases the bandwidth from 115.191 MB/s perceived bandwidth up to 3,122.271 MB/s on the Sierra cluster, representing a $27\times$ increase in write performance. This increase is partially attributable to the use of a separate file per MPI rank, meaning that each file stream is writing stripes to two potentially different servers, making use of a larger majority of the I/O subsystem; the effect this may have on other users of the file system is discussed in the next chapter.

Smaller gains are seen on Minerva, but due to its inferior I/O hardware and GPFS directory level locking, this is to be expected. There are fewer I/O servers to service read and write requests on Minerva and as a result there is much less bandwidth available for the compute nodes.

Figure 5.1 demonstrates that during the execution of BT on 256 cores, concurrent POSIX write calls wait much less time for access to the file system. As each process is writing to its own unique file, it has access to a unique file stream, reducing file system contention. For non-PLFS writes a stepping effect is prominent, where all POSIX writes are queued and complete in a serialised,

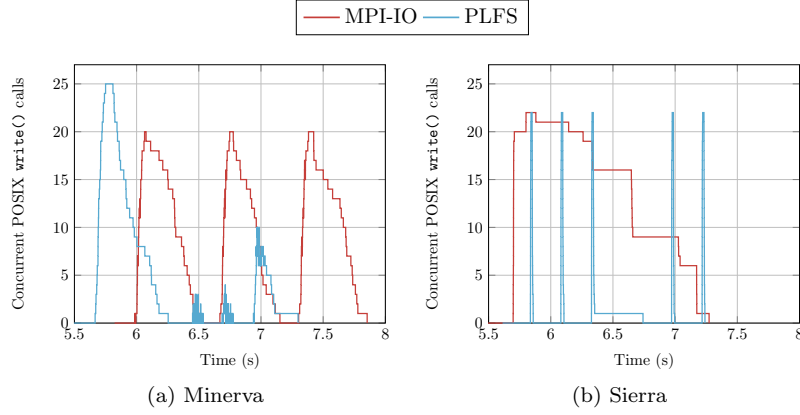


Figure 5.1: Concurrent `write()` operations for BT class C on 256 cores on Minerva and Sierra.

non-deterministic manner. Conversely, on larger I/O installations, PLFS writes do not exhibit this stepping behaviour, and on smaller I/O installations they exhibit this behaviour to a much lesser extent, as the writes are not waiting on other processes to acquire and release file locks.

5.2 Rapid Deployment of PLFS

To reduce the burden of installing and maintaining a PLFS mount point on large production machines, this thesis details the development of an alternative approach to using PLFS. This rapid deployment solution is called ‘LDPLFS’ – as it is dynamically linked (using the Linux linker `ld`) immediately prior to execution, enabling calls to POSIX file operations to be transparently retargeted to PLFS equivalents. Like RIOT, this library requires only a simple environment variable to be exported in order for applications to make use of PLFS – existing compiled binaries, middleware and submission scripts require no modification [143].

This section describes the design and implementation of LDPLFS, showing how a dynamically loadable library can be used to retarget POSIX file operations to PLFS specific file operations. Its performance is assessed on a collection of standard UNIX tools, as well as on three parallel applications running at scale

```
int open(const char *filename, int flags, mode_t mode);
int plfs_open(Plfs_fd *fd, const char *filename, int flags,
             pid_t pid, mode_t mode, Plfs_open_opt *open_opt);

ssize_t write(int fd, const void *buf, size_t count);
ssize_t plfs_write(Plfs_fd *plfsfd, const void *buf,
                  size_t count, off_t offset, pid_t pid);

ssize_t read(int fd, void *buf, size_t count);
ssize_t plfs_read(Plfs_fd *plfsfd, void *buf,
                  size_t count, off_t offset);
```

Listing 5.1: Open, read and write functions from the POSIX and PLFS API.

```
ssize_t read(int fd, void *buf, size_t count) {
    ssize_t ret;

    // check if fd is a plfs file or a normal file
    if (plfs_files.find(fd) != plfs_files.end()) {
        // if the file is a plfs file,
        // find its current virtual offset
        off_t offset = lseek(fd, 0, SEEK_CUR);
        // perform plfs read function
        ret = plfs_read(plfs_files.find(fd)->second->fd,
                        (char *) buf, count, offset);
        // update the virtual offset
        lseek(fd, ret, SEEK_CUR);
    } else {
        // perform a standard read on a normal file
        ret = __real_read(fd, buf, count);
    }
    return ret;
}
```

Listing 5.2: Source code demonstrating POSIX-PLFS translation in LDPLFS.

on the Minerva and Sierra supercomputers. The performance at scale not only demonstrates the applicability of this technique for using virtual parallel file systems, but also demonstrates one of the shortcomings of PLFS.

LDPLFS is a dynamic library specifically designed to interpose POSIX file functions and retarget them to PLFS equivalents. By using the Linux loader, LDPLFS overloads many of the POSIX file symbols (e.g. `open`, `read`, `write`), causing an augmented implementation to be executed at runtime¹. This allows existing binaries and application stacks to be used without the need for recompilation.

¹Although LDPLFS makes use of the `LD_PRELOAD` environmental variable in order to be dynamically loaded, other libraries can also make use of the dynamic loader (by appending multiple libraries into the environmental variable), allowing tracing tools to be used alongside LDPLFS.

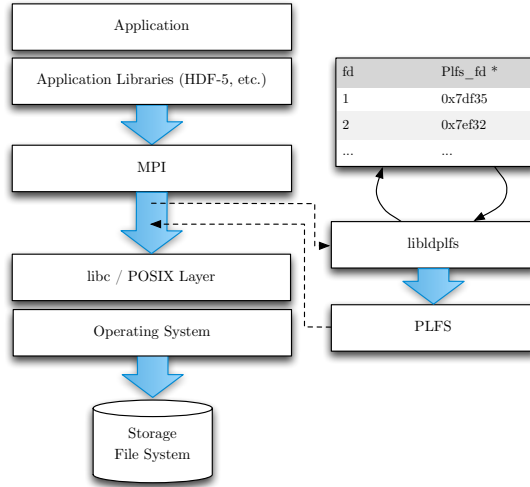


Figure 5.2: The control flow of LDPLFS in an applications execution.

Due to the difference in semantics between the POSIX and PLFS APIs, LDPLFS must perform two essential book-keeping tasks. Firstly, LDPLFS must return a valid POSIX file descriptor to the application, despite PLFS using an alternative structure to store file properties. Secondly, as the PLFS API requires an explicit offset to be provided, LDPLFS must maintain a file pointer for each PLFS file. Listing 5.1 shows three POSIX functions and their PLFS equivalents. Listing 5.2 and the listings in Appendix D show how these POSIX functions can be transparently transformed to make use of the PLFS alternatives.

When a file is opened from within a pre-defined PLFS mount point, a PLFS file descriptor (`Plfs_fd`) pointer is created and the file is opened with the `plfs_open()` function (using default settings for `Plfs_open_opts` and the value of `getpid()` for `pid_t`). In order to return a valid POSIX file descriptor (`fd`) to the application, a temporary file (in our case a temporary file created by `tmpfile()`) is also opened. The file descriptor of the temporary file is then stored in a look-up table and related to the `Plfs_fd` pointer. Future POSIX operations on a particular `fd` will then either be transparently passed onto the POSIX API, or, if a look-up entry exists, passed to the PLFS library.

In order to provide the correct file offset to the PLFS functions, a file pointer

is maintained through `lseek()` operations on the temporary POSIX file descriptor. As demonstrated in Listing 5.2, when a POSIX operation is to be performed on a PLFS container, the current offset of the temporary file is established (through a call to `lseek(fd, 0, SEEK_CUR)`), a PLFS operation is performed (again using `getpid()` where needed), and then finally, the temporary file pointer is updated (once again through the use of `lseek()`). Figure 5.2 shows the control flow of an application when using LDPLFS.

5.2.1 Performance Analysis

Feasibility Study

The initial assessment of LDPLFS was conducted on Minerva. The MPI-IO Test application from LANL was used to write a total of 1 GB per process in 8 MB blocks [95]. Collective blocking MPI-IO operations were employed with tests using PLFS through the FUSE kernel library, the `ad_plfs` MPI-IO driver and LDPLFS. In all cases the OpenMPI library used was version 1.4.3 with PLFS version 2.0.1. The performance results were then compared to the achieved bandwidth figures from the default `ad_ufs` MPI-IO driver without PLFS.

Tests were conducted on between 1 and 64 compute nodes using 1, 2 and 4 cores per node²³. Each run was conducted with collective buffering enabled and in the default MPI-IO configuration⁴ in order to provide better performance with minimal configuration changes. The node-wise performance should remain largely consistent, while the number of cores per node is varied – in each case there remains only one process on each node performing the file system write. As the number of cores per node is increased, an overhead is incurred because of the presence of on-node communication and synchronisation.

Figure 5.3 demonstrates promising results, showing that the performance

²Due to machine usage limits, using all 12 cores per node would limit the results to a maximum of 16 compute nodes, decreasing the scalability of the results.

³In some cases, other jobs were present on the compute nodes in use. Full numeric data along with the 95% confidence intervals are given in Appendix E

⁴The default collective buffering behaviour is to allocate a single aggregator per *distinct* compute node.

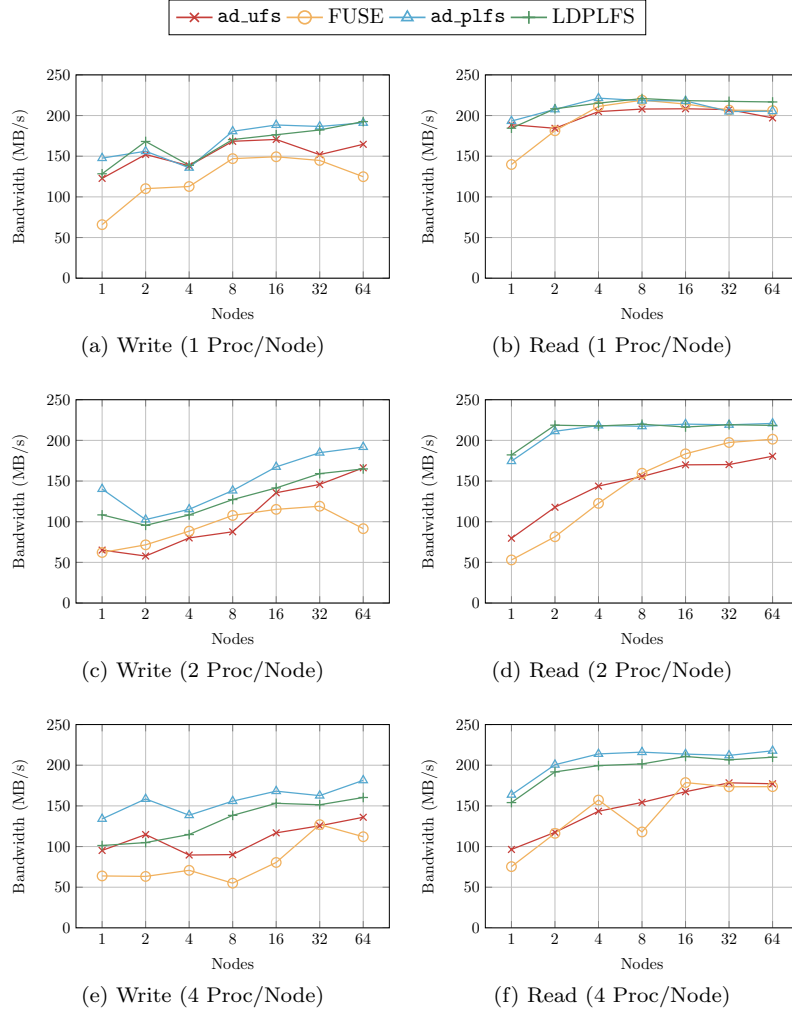


Figure 5.3: Benchmarked MPI-IO bandwidths on FUSE, the `ad_plfs` driver, LDPLFS and the standard `ad_ufs` driver (without PLFS).

of LDPLFS closely follows the performance of PLFS through ROMIO and is significantly better than FUSE (up to $2\times$) in almost all cases. It is interesting to note that on occasion, LDPLFS performs better than the `ad_plfs` MPI-IO driver; however as can be seen from the confidence intervals, this is largely an artefact of machine noise (numerical data can be found in Appendix E). On Minerva, the performance of FUSE is worse than standard MPI-IO by 20% on average for parallel writes. FUSE is known to degrade performance, due to additional memory copies and extra context switches [70], and while this

	PLFS Container		Standard UNIX File	
<code>cp</code> (read)	100.713	(97.182, 103.949)	114.279	(110.878, 119.906)
<code>cp</code> (write)	107.587	(106.473, 110.473)		
<code>cat</code>	25.186	(23.413, 26.155)	25.433	(22.548, 28.469)
<code>grep</code>	130.662	(127.643, 131.396)	128.863	(126.445, 129.093)
<code>md5sum</code>	26.970	(26.018, 27.035)	26.781	(25.612, 28.511)

Table 5.2: Time in seconds for UNIX commands to complete using PLFS through LDPLFS, and without PLFS.

overhead is addressed by Bent et al. [11], the I/O set-up used in that study is much larger than that used by Minerva, and makes use of custom optimised hardware. It is therefore likely that the much greater performance increase generated by PLFS masks this overhead much better than the I/O hardware of Minerva.

Standard UNIX Tool Performance

One of the current difficulties associated with the practical use of PLFS is the complexity of managing PLFS containers. Since FUSE treats a PLFS mount point as a self-contained file system, using the files in any application is trivial. However, when using either of the alternative solutions for PLFS, applications must either use MPI or be rewritten for PLFS. Files created under a PLFS mount point appear inside the “backend” directory as directories themselves with hundreds of files (see Figure 3.9 in Chapter 3). Visualising data or post processing the information output becomes difficult in this scenario; this is one of the problems LDPLFS additionally addresses. As LDPLFS operates at the POSIX call level, it can be used with any standard UNIX tools as well as parallel science and engineering applications.

Table 5.2 presents the performance of several standard UNIX tools operating on a 4 GB PLFS file container. Note that the file copy (`cp`) times correspond to copying a file from a PLFS container to a standard UNIX file and vice versa. These can be compared to a single time for copying from and to a standard UNIX file.

Since each of these commands are serial applications, each command was

executed on the login node of Minerva. It is promising to see that the time for each of the commands to complete is largely the same for both standard UNIX files and PLFS container structures. These results show that PLFS is marginally faster when copying to or from a PLFS file than a normal UNIX file. This improvement in performance may be attributed to the increased number of file streams available, improving the bandwidth achievable from the file servers.

The results presented above position LDPLFS as a viable solution to improving the performance of I/O in parallel, as well as showing that there is no substantial performance hit when using LDPLFS to interact with PLFS mount points using serial (non-MPI) applications. Using LDPLFS, it is now much easier to assess the performance of PLFS on a variety of systems without the overhead associated with compiling a customised MPI library, or requesting access to the FUSE kernel driver. In the next section, the performance of LDPLFS at much larger scales is demonstrated using a small set of I/O intensive mini-applications.

Parallel Application Performance

Figure 5.3 shows that on Minerva, PLFS improves performance by approximately $2\times$ for parallel applications. Because of the relatively small I/O set-up employed by Minerva, achieving performance increases such as those reported by Bent et al. [11] – where a high-end PanFS I/O solution is used – is most likely not possible. In order to better demonstrate how PLFS and LDPLFS perform on a much more substantial I/O set-up, two applications have been used to benchmark the *lscratchc* file system attached to Sierra (see Table 3.3 in Chapter 3).

For this study the previously introduced BT solver and FLASH-IO have been used. For BT, problem class C ($162 \times 162 \times 162$) has been used, writing a total of 6.4 GB of data during an execution, as well as the D problem class ($408 \times 408 \times 408$), writing a total of 136 GB of data. The application is strong scaled – as the number of cores is increased, the global problem size remains

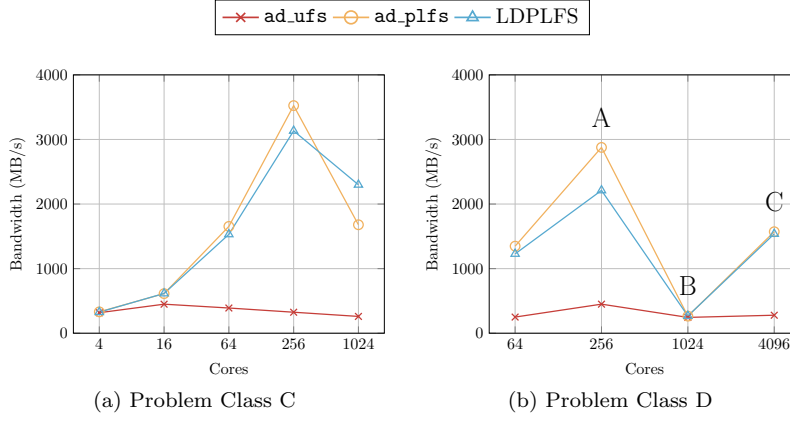


Figure 5.4: BT benchmarked MPI-IO bandwidths using MPI-IO, as well as PLFS through ROMIO and LDPLFS.

the same, with each process operating over a smaller sub-problem. For the C problem class, the global problem size is relatively small, and can only be scaled to 1,024 cores before the local problem size becomes too small to operate on correctly. Conversely for the D program class, the global problem size is so large that on less than 64 cores, the execution time becomes prohibitive. For this reason this study employs between 4 and 1,024 cores for the C problem class, and between 64 and 4,096 cores for the D problem class.

Figures 5.4 and 5.5 show the achieved bandwidth for the two mini-applications in their default configurations using the system’s pre-installed OpenMPI version 1.3.4 library (`ad_ufs`); with the system’s OpenMPI library and LDPLFS; and finally with the MPI-IO PLFS file system driver (`ad_plfs`) compiled into a customised build of OpenMPI (version 1.4.3). The performance of PLFS through the two methods is largely the same, with a slight divergence at scale – as previously, full numerical data can be found in Appendix E.

Since LDPLFS retargets POSIX file operations transparently and uses various structures in memory to maintain file consistency, a change in the local problem size may affect the LDPLFS performance due to the memory access patterns changing and additional context switching. Furthermore, write caching can produce a large difference in performance – where data is small enough to

fit in cache, the writing of that data to disk can be delayed.

Write caching is most prevalent in the BT application where, at large-scale, small amounts of data are being written by each process during each parallel write step. For the C problem class (Figure 5.4(a)), 6.4 GB of data is written in 20 separate MPI write calls, causing approximately 300 KB of data to be written by each process at each step. When writing to a single file, the file server must make sure that writes are completed before allowing other processes to write to the file. This causes each write command to wait on all other processes, leading to relatively poor performance. Conversely, through PLFS, each process writes to its own file, thereby allowing the write to be cleared to cache almost instantaneously.

In Figure 5.4(b), the performance peaks at nearly 3000 MB/s (point A) due to the increased parallelism exposed by PLFS. The performance then rapidly decreases at 1,024 cores (point B), where each process is writing approximately 136 MB, in 20 steps. It is likely that these writes (of approximately 7 MB each) are marginally too large for the system's cache and therefore must be written to disk. This potentially creates a large amount of contention on the file system, causing performance that is equivalent to vanilla MPI-IO. However, when using 4,096 cores, each write is less than 2 MB per process, writing only 34 MB per process during the execution (point C). This causes the write-caching effects seen in Figure 5.4(a) to reappear.

FLASH-IO is a weak scaled problem, and for these experiments the local problem size was set to $24 \times 24 \times 24$. This causes each process to write approximately 205 MB to the disk, through the HDF-5 library [73]. Runs were conducted on between 1 node and 256 nodes, using all 12 cores each time, thus using up to 3,072 cores. Note that as the number of compute nodes is increased, so too is the output file size. Since each process was writing the same total amount of data, over the same number of time steps, caching effects were less prevalent for these weak scaled problems.

Interestingly, Figure 5.5 shows that as the core count is increased on FLASH-

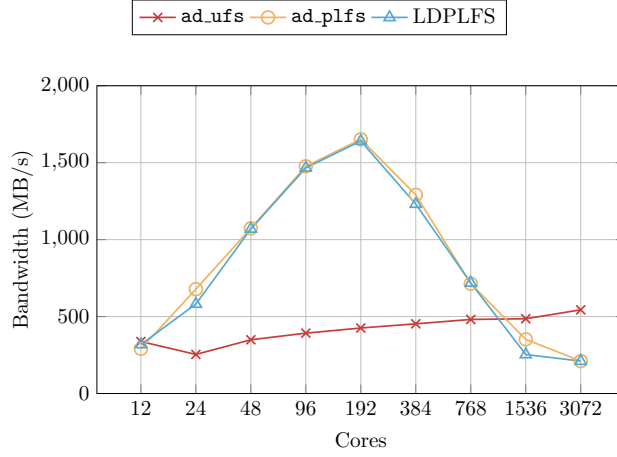


Figure 5.5: FLASH-IO benchmarked MPI-IO bandwidths using MPI-IO, as well as PLFS through ROMIO and LDPLFS.

IO, the write speed of MPI-IO gently increases up to approximately 550 MB/s. However, when using PLFS a sharp increase in write speed is demonstrated up to 192 cores (or 16 nodes), at which point the average write speed reaches approximately 1,650 MB/s, before decreasing to 210 MB/s at 3,072 cores.

This decrease in performance may be explained by contention on the MDS; the Lustre file system uses a dedicated MDS and therefore, as the number of cores is increased, the performance may plateau and then decreases due to the MDS becoming a bottleneck in the system. Since PLFS operates using multiple files per core (at least one for the data and one for the index), it uses many more files as the problem is scaled, potentially putting a large load on the MDS. This bottleneck would be less evident in the BT mini-application due to the small write sizes. However, without direct access to the MDS this theory is difficult to explore fully.

Another explanation for the decrease in performance at scale is that, as many files are created and each single file is striped across two servers (in the default case), at larger core counts data is split into a large number of stripes ($2 \times \#$ of files). Creating more stripes than there are OSTs may introduce large overheads on the file servers and may increase contention (as many of the files will be striped across shared OSTs). This is investigated further in Chapter 6.

5.3 Summary

File I/O operations have, in many cases, been one of the last aspects considered during application optimisation. In this chapter the performance gains of PLFS have been partially explored and LDPLFS, which offers the opportunity to accelerate file read and write activities without modification to the machine's environment or an application's source code, has been presented.

Specifically, it has been shown that PLFS creates many more file streams and may therefore avoid file system contention by removing the need for file locks. However, at large scale, PLFS may actually be detrimental to performance by creating more file system stripes than there are servers.

In order to address the issue of installation and maintenance of PLFS, the development of a dynamically loadable library has also been presented that allows users to assess the benefits of PLFS without the installation burden. The performance of LDPLFS has been compared to PLFS using the FUSE Linux kernel module, PLFS using the MPI-IO file system driver and the original MPI-IO library without PLFS. In this comparison LDPLFS was able to offer approximately equivalent performance to using PLFS through the MPI-IO file system driver and improved performance over FUSE.

LDPLFS is a solution which requires only the PLFS library and itself to be built with no system administrator actions, thus forgoing the need to install FUSE or a custom MPI library. The library is loadable from only a single environmental variable, yet is potentially able to offer a significant improvement in parallel I/O activity. The work presented in this chapter may be useful to several industry partners, as such a solution helps to address concerns which may arise over the security model of FUSE and the significant investment associated with the recompilation of applications using a custom MPI-IO middleware. LDPLFS therefore straddles the gap between offering improved application performance and the effort associated with the installation of traditional PLFS.

CHAPTER 6

Parallel File System Performance Under Contention

The optimisation of I/O performance has largely been the responsibility of application developers, configuring their own software to achieve the best performance – a responsibility that has often been ignored. Software solutions to achieving better performance, such as custom-built MPI-IO drivers that target specific file systems, are often not installed by default. For instance, on the systems installed at the Lawrence Livermore National Laboratory (LLNL), an optimised Lustre-specific driver is not installed despite the software being available as standard in most MPI libraries.

The experiments described so far in this thesis have been performed using the system “stock” MPI installations, or – in the case of Chapter 5 – a version of MPI compiled with PLFS built in. In this chapter, an MPI library with an optimised Lustre driver is built and utilised on the Cab machine (see Section 3.4 for details) at LLNL. Research by Behzad et al. [9, 10], Lind [76] and You et al. [148] suggests that performance can be improved by as much as two orders of magnitude with the correct settings and the Lustre-optimised driver.

In this chapter a parameter sweep is used to search for an optimised configuration for a small test with IOR. Through this, a performance improvement of $49\times$ is shown. However, while performance is vastly improved by adjusting the Lustre settings, optimal performance for one application can reduce the quality of service (QoS) provided to other users on a shared system such as Cab. The effect of OST contention on I/O intensive workloads is demonstrated, showing that using optimised settings for four competing tasks may result in a $3-4\times$ reduction in performance for each application. Reducing the number of requested resources increases the system’s QoS, at minimal cost to the overall performance

Option	Value
API	MPI-IO
Write file	On
Read file	Off
Block Size	4 MB
Transfer Size	1 MB
Segment Count	100

Table 6.1: IOR configuration options for experiments.

of the four tasks.

Finally, the optimised Lustre configuration is compared to the performance of PLFS on the same file system. Below 512 processes, PLFS boosts performance over that which can be achieved with an optimal Lustre configuration; however, at scale, PLFS becomes detrimental to performance, just as was the case in the previous chapter.

6.1 Effective Use of Uncontended Parallel File Systems

As demonstrated in Chapters 4 and 5, the large parallel file systems connected to some of the most powerful supercomputers in the world are currently being under utilised – partially due to a lack of available software drivers, partially due to lack of optimisation in the applications themselves. The work presented by Behzad et al. [10] shows how using the Lustre-specific MPI-IO driver (`ad_lustre`) distributed with most MPI implementations can lead to performance improvements of up to 100× over the default installation. The authors use a genetic algorithm to search the parameter space for an optimised configuration [10], varying the stripe factor (the number of OSTs to use), the stripe size, the number of collective buffering nodes and the collective buffer size (as well as some HDF-5 specific options).

In this thesis a small IOR problem is configured in order to demonstrate the benefits and consequences of this form of performance tuning. IOR was configured such that each process wrote 100 4 MB blocks to a file in chunks of

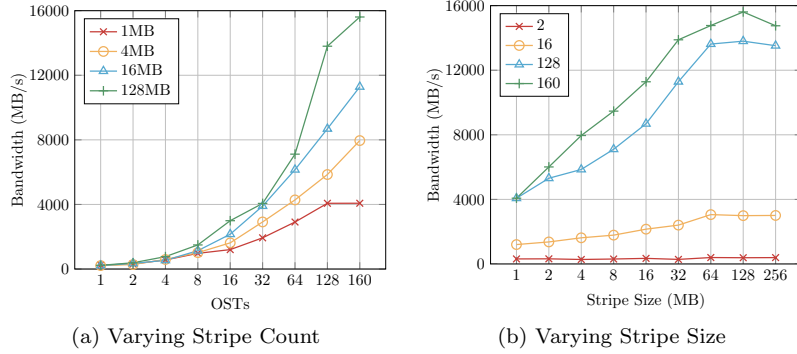


Figure 6.1: Write bandwidth achieved over 1,024 cores by varying just the stripe count and just the stripe size.

1 MB. The parameters were chosen such that each write matched the default stripe size, providing the best possible performance in Lustre’s default configuration. In order to find an optimised Lustre configuration, a parameter sweep was performed on 64 nodes ($64 \times 16 = 1,024$ cores). The collective buffer size was set to the default value (16 MB) and each node contributed one collective buffer process, meaning there was a total of 64 buffering processes. To reduce the search space, a linear search was conducted with a stripe count between 1 and 160 (as there is a 160 OST limit in the Lustre version 2.4.0, which is used on OCF machines) and a stripe size between 1 and 256 MB.

Selected results of this parameter search are shown in Figures 6.1 and 6.2 (additional numeric data can be found in Appendix F). Using the default Lustre configuration (stripe count = 2, stripe size = 1 MB), the application achieves an average of 313 MB/s. Through varying the stripe size, performance can be increased from this baseline bandwidth up to 395 MB/s, and through varying the stripe count performance can be increased much further, up to a maximum of 4,075 MB/s.

Figure 6.2 shows that through varying both parameters the maximum bandwidth is found when using 160 stripes of size 128 MB; performance increases from the baseline 313 MB/s, up to 15,609 MB/s, representing a $49\times$ improvement in write performance. This result largely echoes previous work, where the

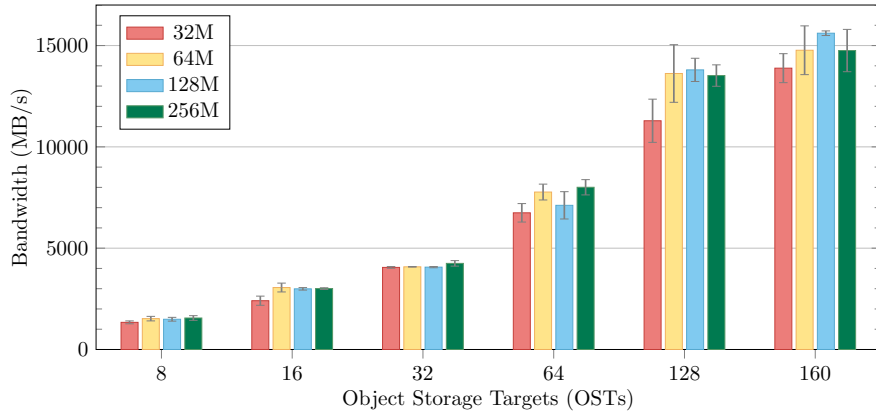


Figure 6.2: Write bandwidth achieved over 1,024 cores by varying both the stripe count and the stripe size.

greatest performance is usually found by striping across the maximum number of OSTs and writing stripes that are a multiple of the application’s I/O block size [10, 76, 148].

That the optimal performance is found when exploiting the maximum amount of available parallelism may seem obvious, but on many systems, it is simply not possible to achieve this without a rebuilt software stack. Moreover, there are a finite number of available file servers and storage targets; exploiting a larger proportion of these may be optimal on a quiet system, but when many tasks require I/O simultaneously, the performance may decrease due to OST contention.

6.2 Quantifying the Performance of Contended File Systems

On a multi-user system, with limited resources, using a large percentage of the OSTs available may be detrimental to the rest of the system. The *lscratchc* file system used in this chapter exposes 480 OSTs to the user¹. The assignment of OSTs to files is performed at file creation time, with targets assigned at random

¹The *lscratchc* file system was upgraded from 360 OSTs to 480 OSTs between the experiments in Chapters 4 and 5, and the experiments in this chapter

(based on current usage, to maintain an approximately even capacity). This suggests that three jobs using 160 OSTs each would fully occupy the file system, if the assignment guaranteed no overlaps. However, as OSTs are assigned randomly, for two jobs (of which the first uses 160/480 of the available OSTs) approximately one third of the OSTs assigned to the second job will be on OSTs in use by the first job.

$$D_{\text{inuse}}(n) = D_{\text{inuse}}(n-1) + \left(r_j - \frac{D_{\text{inuse}}(n-1)}{D_{\text{total}}} r_j \right) \quad (6.1)$$

If each job ($j \in \{1 \dots n\}$) requests r_j OSTs, the total number of OSTs in use (D_{inuse}) after each job starts is described by Equation 6.1, where $D_{\text{inuse}}(0) = 0$. Each time a new job starts, the number of OSTs in use increases by the size of the request, minus the average number of OST collisions that occur. If each job is requesting the same number of resources (R) – which may be the case if a parameter sweep has determined that the optimal configuration is when the maximum number of OSTs are used – then the number of OSTs in use can be simplified to:

$$D_{\text{inuse}} = D_{\text{total}} - \left(D_{\text{total}} \times \left(1 - \frac{R}{D_{\text{total}}} \right)^n \right) \quad (6.2)$$

With these two equations the average load of each OST (D_{load}) can be calculated, for any particular workload, by taking the number of stripes requested in total, and dividing it by the number of OSTs in use. A load of 1 would imply that each OST is only in use by a single job, whereas a higher number would indicate that there are a number of collisions on some OSTs, potentially resulting in a job switching overhead.

$$D_{\text{req}} = R \times n \quad (6.3)$$

$$D_{\text{load}} = \frac{D_{\text{req}}}{D_{\text{inuse}}} \quad (6.4)$$

$D_{\text{total}} = 480, R = 160$				$D_{\text{total}} = 480, R = 64$			
Jobs	D_{inuse}	D_{req}	D_{load}	Jobs	D_{inuse}	D_{req}	D_{load}
1	160.000	160	1.000	1	64.000	64	1.000
2	266.667	320	1.200	2	119.467	128	1.071
3	337.778	480	1.421	3	167.538	192	1.146
4	385.185	640	1.662	4	209.199	256	1.224
5	416.790	800	1.919	5	245.306	320	1.304
6	437.860	960	2.192	6	276.599	384	1.388
7	451.907	1120	2.478	7	303.719	448	1.475
8	461.271	1280	2.775	8	327.223	512	1.565
9	467.514	1440	3.080	9	347.593	576	1.657
10	471.676	1600	3.392	10	365.247	640	1.752

Table 6.2: The average number of OSTs in use and their average load based on the number of concurrent I/O intensive jobs.

Table 6.2 demonstrates this for the *lscratchc* file system where each job is requesting the previously discovered optimal number of stripes, 160, and when that request is reduced to 64. With 10 simultaneous I/O intensive jobs, an average of 4 collisions will occur on each OST, though a small subset of OSTs may well incur all 10 potential collisions (and some may incur none), reducing the performance of the file system for every job. By reducing the size of the stripe requests, the load is reduced, possibly avoiding many of the bottlenecks associated with OST contention.

In order to ascertain how the OSTs in the *lscratchc* file system behave under contention, a study was undertaken using a custom-written benchmark that creates a split communicator, allowing each process to read and write to its own file in a single MPI application. The benchmark then opens a number of files, with the same Lustre configuration (a single 1 MB stripe). Using the `stripe_offset` MPI hint, the OST to use is specified such that every rank writes to a file striped on the same target. Figure 6.3 shows the per-process bandwidth achieved with a varying number of contended file writes.

Figure 6.3 shows the average per-task performance for concurrent tasks writing to the same OST simultaneously. The shaded area represents the estimated performance as calculated from the single job experiment’s 95% confidence intervals and scaled linearly; as *lscratchc* is already a shared-user file system, there is some variance in performance with no *forced* contention. The graph shows that, as the number of jobs is increased, the performance diverges from

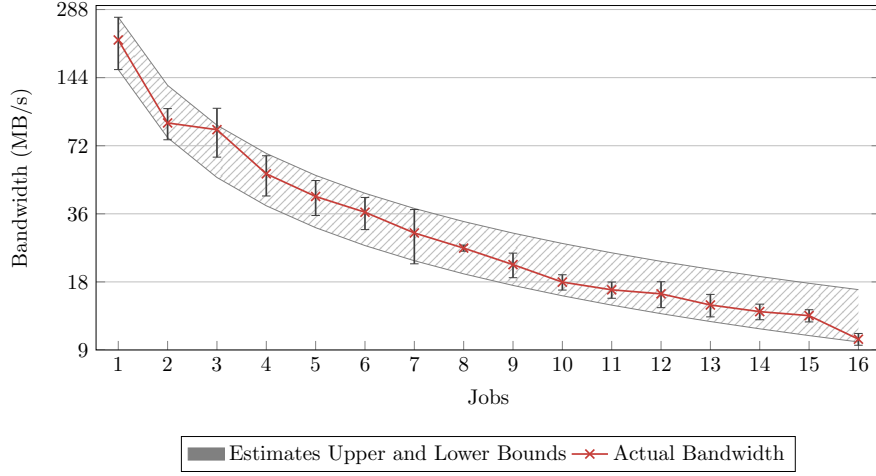


Figure 6.3: The performance per-task of the *lscratchc* file system under contention, with the ideal upper and lower bounds.

the top of the ideal scaling line, illustrating the overhead associated with task serialisation.

To investigate this more thoroughly, a job was submitted to Cab that created four identical IOR executions each running simultaneously with the configuration stated in Table 6.1. Each task used 64 nodes (1,024 processes) and thus the total job consumed 4,096 cores, and the MPI hints were specified according to the previously discovered optimal values (Figure 6.2). As can be seen in Figure 6.4, each individual application achieved approximately 4,500 MB/s – a $3.44\times$ reduction from the peak value (15,609 MB/s) seen in Figure 6.2.

Using the mean of five experiments, Table 6.3 and Figure 6.5 demonstrate how reducing the number of stripes per job increases the OST availability to the rest of the system while having a minimal effect on performance. Using as few as 32 stripes per file, the average bandwidth achieved by each of the four applications is 3,500MB/s, but by Equation 6.2, only 115 OSTs will be in use in the average case, providing an average OST load of ≈ 1.1 .

Furthermore, Table 6.3 shows that when using a stripe count of 160, there are 42 OSTs that are being contended by 3 of the 4 jobs and there are 7 OSTs being contended by all 4. By reducing the demand to 64 stripes, the performance is

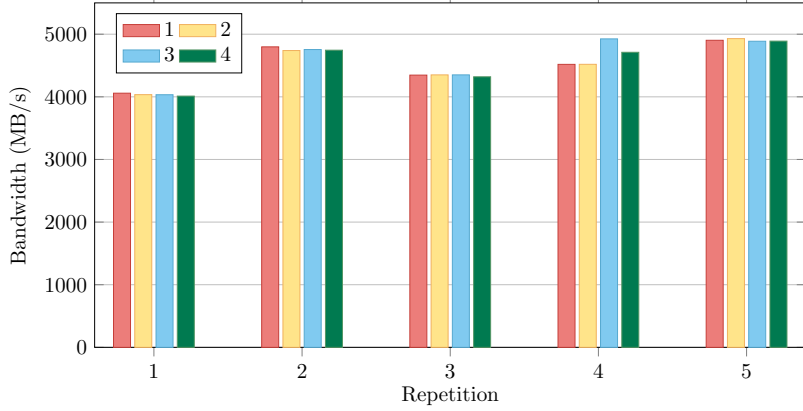


Figure 6.4: Performance of each of 4 tasks over 5 repetitions where all tasks are contending the file system.

R	Average	Total	D_{req}	OST Usage				Predicted		Actual	
	Bandwidth	Bandwidth		1	2	3	4	D_{inuse}	D_{load}	D_{inuse}	D_{load}
32	3654.061	14616.244	128	103.2	11.2	0.8	0.0	115.759	1.106	115.200	1.111
64	3910.507	15642.026	256	172.6	35.8	3.4	0.4	209.199	1.224	212.200	1.207
96	4042.980	16171.918	384	199.4	76.4	9.8	0.6	283.392	1.355	286.200	1.342
128	4172.166	16688.662	512	211.6	111.4	22.4	2.6	341.182	1.501	348.000	1.472
160	4541.366	18165.462	640	191.8	147.0	41.8	7.2	385.185	1.662	387.800	1.650

Table 6.3: Average and total bandwidth achieved across four tasks for a varying stripe size request, along with values for the average number of tasks competing for 1, 2, 3 and 4 OSTs respectively.

reduced by $\approx 14\%$ while the number of OSTs in use is reduced by $\approx 37\%$, leaving more resources available for a larger number of tasks, while also reducing the number of collisions significantly.

Although the optimal performance on *lscratchc*, with four competing tasks, is still found using the maximum number of OSTs allowed, the bandwidth achieved is almost a quarter of the previously achieved maximum. On file systems where there are less OSTs (such as those used by Behzad et al. [10]), any job contention will decrease the achievable performance and may be detrimental to the rest of the system. To demonstrate this further the equations presented in this thesis have been applied to the configuration of the Stampede supercomputer described in [10]. Table 6.4 shows the predicted OST load for Stampede’s file system using the optimal stripe count found by Behzad et al. for the VPIC-IO application (128 stripes on a file system with 58 OSSs and 160 OSTs). Table 6.4

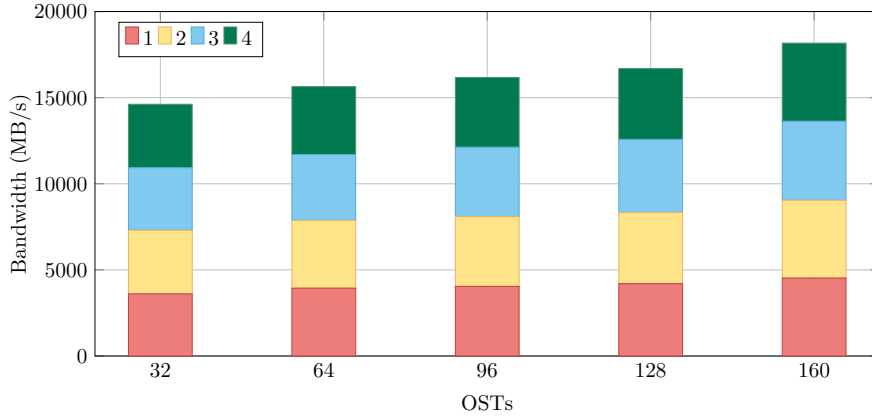


Figure 6.5: Graphical representation of the data in Table 6.3, showing optimal performance at 160 stripes per file, but very minor performance degradation at just 32 stripes per file.

$D_{\text{total}} = 160, R = 128$			
Jobs	D_{inuse}	D_{req}	D_{load}
1	128.000	128	1.000
2	153.600	256	1.667
3	158.720	384	2.419
4	159.744	512	3.205
5	159.949	640	4.001
6	159.990	768	4.800
7	159.998	896	5.600
8	160.000	1024	6.400
9	160.000	1152	7.200
10	160.000	1280	8.000

Table 6.4: OST usage and average load for the Stampede I/O setup described by Behzad et al. [10].

demonstrates that with only three equivalent simultaneous tasks with a similar I/O demand, the OST load figure suggests that the majority of the OSTs are being used by as many as two or three simultaneous tasks.

6.3 Performance Comparison: Lustre vs. PLFS

In the previous chapter, PLFS was shown to produce a noticeable performance increase on LLNL systems under certain conditions. However, this thesis has already demonstrated that the performance gap is reduced when using the optimised MPI-IO driver. Furthermore, the results in the previous chapter (Figure 5.5) show that at scale, PLFS performs worse than even the unopti-

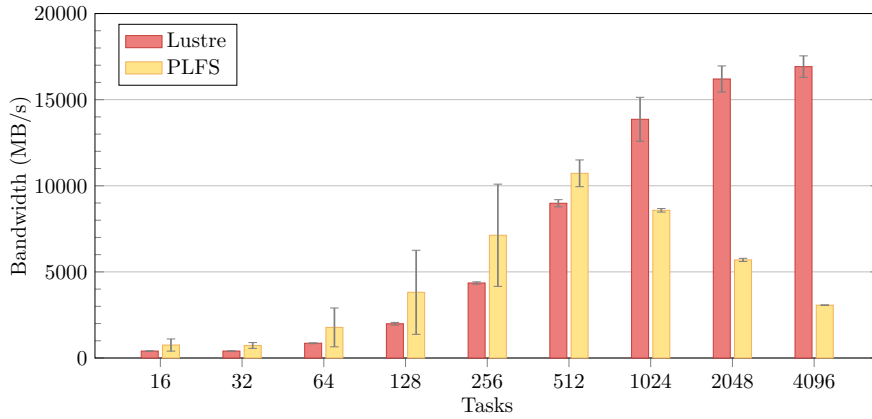


Figure 6.6: Achieved write bandwidth achieved for IOR through an optimised Lustre configuration and through the PLFS MPI-IO driver.

mixed UNIX file system (UFS) MPI-IO file system driver (`ad_ufs`).

Figure 6.6 shows the performance of the *lscratchc* file system when running the IOR problem described in Table 6.1 on Cab. PLFS operates by creating a separate data and index file for each rank, in directories controlled by a hashing function; this increases the number of file streams available and consequently increases the number of Lustre OSTs in use. As the files are written by PLFS through POSIX file system calls, each file is created with the system default configuration of two 1 MB stripes per file (unless otherwise specified using the `lfs` control program).

It should be noted that as PLFS creates a large number of files, with randomly placed stripes, there is a larger variance in PLFS performance. An execution running with 256 processes will create 256 data files, requiring 512 stripes. Experimentally, this produces an average OST load of 1.58 and a bandwidth between 3,329.9 MB/s and 11,539.4 MB/s (average 7,126.9 MB/s). Conversely, through the Lustre driver, the variance is much lower as at most 160 stripes will be created with no collisions between OSTs. Due to background machine noise it is difficult to know what the load is on each OST at any given time, but generally PLFS performs better when the number of OSTs experiencing a high number of collisions is minimised.

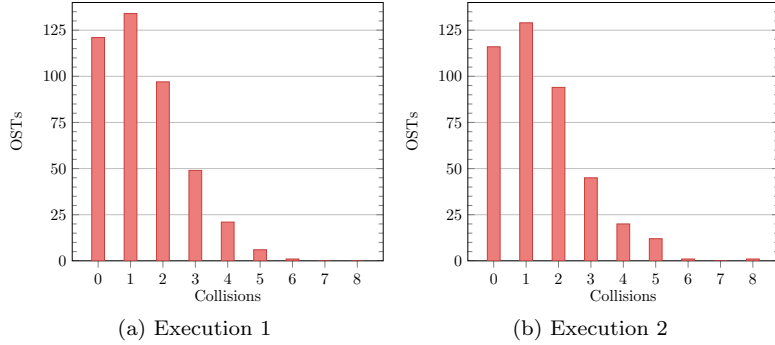


Figure 6.7: The number of OST collisions for IOR running through PLFS with 512 cores.

Figure 6.7 shows the number of OST collisions in the PLFS backend directory for the 512 cores case, for two of the five experiments. At 512 cores, the performance of PLFS reaches its peak before Lustre begins to provide better performance. Equation 6.2 can be amended to work for PLFS by treating each rank as a separate task with 2 stripes (i.e. $R = 2$) and setting the number of tasks (n) to the number of ranks in use.

$$D_{\text{inuse}} = D_{\text{total}} - \left(D_{\text{total}} \times \left(1 - \frac{2}{D_{\text{total}}} \right)^n \right) \quad (6.5)$$

$$D_{\text{load}} = \frac{2 \times n}{D_{\text{inuse}}} \quad (6.6)$$

With this in mind, it becomes an inevitability that on a reasonable Lustre file system, PLFS provides a small-scale fix, but overwhelms the file system at higher core counts. Using Equations 6.5 and 6.6, at 512 cores on the *lscratchc* file system, there is an average of 2.4 tasks using each OST, by 688 cores, there are 3 tasks per OST (which is shown in Figure 6.3 to still provide “good” performance); at 2,048 and 4,096 cores, the number of collisions reaches 8.53 and 17.06 respectively, which begins to saturate the file system and decreases performance not just for the host application, but for all other users of the file system too. Table 6.5 and Appendix G shows the numeric collision statistics

Collisions	Experiment				
	1	2	3	4	5
0	0	0	0	0	0
1	0	0	0	0	0
2	1	0	0	0	0
3	0	0	1	0	0
4	0	0	0	0	0
5	0	1	1	1	0
6	1	2	2	2	4
7	2	4	2	10	2
8	8	7	5	3	7
9	9	10	13	16	15
10	15	13	21	18	18
11	26	18	30	21	25
12	33	38	34	37	29
13	48	46	36	33	37
14	45	48	38	40	48
15	28	33	45	51	46
16	51	49	32	44	46
17	42	42	46	41	36
18	30	35	34	29	33
19	44	46	39	34	29
20	28	21	27	20	25
21	24	18	21	22	26
22	17	14	14	12	17
23	10	9	14	11	10
24	6	12	4	8	9
25	1	3	8	11	7
26	5	5	8	9	5
27	4	5	4	3	2
28	0	0	1	1	3
29	2	1	0	2	0
30	0	0	0	0	0
31	0	0	0	0	0
32	0	0	0	1	0
33	0	0	0	0	0
34	0	0	0	0	0
35	0	0	0	0	1
D_{inuse}	480	480	480	480	480
D_{load}	17.07	17.07	17.07	17.07	17.07
BW (MB/s)	3042.06	3077.16	3083.26	3084.89	3057.90

Table 6.5: Stripe collision statistics for PLFS backend directory running with 4,096 cores.

for other application configurations. Table 6.5 specifically shows that at 4,096 cores, most OSTs are being used for the stripe data of between 10 and 23 files and in one of the five repeated experiments, a single OST is being used by 35 competing ranks, placing a large overhead on its potential performance.

6.4 Summary

In this chapter it has been shown that current-day I/O systems perform much better than some literature suggests [11, 24, 29]. However, this level of performance can only be achieved if the file system is being used correctly. Behzad et

al. [9,10] suggest that on Lustre file systems, a good level of performance can be found and that up to a point, the file system scales almost linearly. However, due to restrictions in the version of Lustre used for the experiments in this thesis, the maximum number of OSTs that can be used is only 160; this suggests that when problems are scaled up to millions of nodes (as may be the case for exascale), the I/O performance will not scale. Particular versions of Lustre already scale beyond this OST limit [41], but they are not currently being used by some of the biggest supercomputing centres (such as the OCF at LLNL).

In this chapter an exhaustive search algorithm has been used to perform a parameter sweep to find a more performant configuration for the Lustre file system connected to the Cab supercomputer. After finding such a configuration for a small IOR problem, the effect of I/O contention on the achievable bandwidth is analysed. This chapter uses a small problem to show the effect of contention on an easily scalable job. This analysis therefore demonstrates what happens in the best possible case, showing that even well aligned jobs are affected heavily by contention. A series of metrics have been proposed that capture the contention that is created by several jobs competing for a shared resource. Section 6.2 shows that when jobs are run with an optimised configuration on a contended resource, performance drops considerably, and using fewer resources vastly improves system availability with a minor performance degradation.

The previous chapter explored the work of LANL and EMC² in creating PLFS, which has been shown to provide significant speed-ups at medium-scale. However, this thesis has demonstrated that PLFS may be harmful to performance on Lustre file systems at large-scale. The work in this chapter provides an explanation of this phenomenon within the framework of the provided Lustre contention metrics.

Using the equations given, the load of each OST can be calculated for both competing I/O intensive applications and for PLFS-based applications. With the results from these equations, various file system purchasing decisions can be

made; for instance, the number of OSTs can be increased in order to reduce the OST load for a theoretically “average” I/O workload. Furthermore, the benefits PLFS may have on an application can be calculated based on the scale at which it will be run, as well as on the number of OSTs available for the task.

While, at the time of writing, the I/O backplanes in modern day systems are being under-utilised, with the correct configuration options and some optimisation by application developers, acceptable performance can be achieved with relatively little effort. Making these changes to applications and file system configurations will not only improve current scientific applications, but will also benefit future systems and inform future I/O system developers on how to best proceed towards exascale-class storage.

CHAPTER 7

Discussion and Conclusions

The work presented in this thesis outlines the current and potential future state of I/O in parallel applications. As supercomputing approaches billion-way parallelism [119], node failures will become more prevalent; failure resilience will be required to ensure computation will continue almost uninterrupted. The usual approach of checkpointing may not be sufficient at exascale to recover from failures [19, 44]. However, many science codes have settled code-bases and will therefore be resistant to significant changes in the way resilience is maintained. Additionally, checkpointing often provides additional benefits, such as sub-problem analysis and visualisation. For this reason, work in improving storage systems is still necessary if exascale computing is to become a reality [105].

Much of the literature on improving parallel I/O is focused on changing the way current file systems operate, to perform write operations in a way that is more conducive to spinning hard disks, but many of the comparisons presented are against popular file systems operating in an unoptimised fashion [11]. As demonstrated by Behzad et al. [10] and Hedges et al. [63], much better performance can come from traditional parallel file systems like Lustre and IBM's General Parallel File System (GPFS). Often, large parallel systems are configured to provide "acceptable" I/O performance for a large number of concurrent general-purpose applications but, as demonstrated in this thesis, better performance can be achieved without negatively impacting global availability, if the workload of a parallel machine is well understood.

Specifically, Chapter 4 has demonstrated that by analysing the I/O behaviour of parallel codes, the large overhead associated with MPI and file-formatting libraries can be reduced by selecting options that are more suitable

for an application’s I/O patterns. Through analysis with the RIOT I/O toolkit, the inefficiencies of *data-sieving* were identified in two HDF-5 based applications where, through adjusting the MPI-IO driver options, performance was improved by eliminating the “lock, read, write, unlock” cycles that data-sieving induces. By analysing the information gathered by RIOT these bottlenecks were identified and subsequently performance was improved by at least $2\times$.

Chapter 5 further demonstrated the applicability of RIOT to analysing the underlying behaviour of parallel file systems. Specifically, the gains reported by Bent et al. were investigated, showing that at low core counts, the parallel log-structured file system (PLFS) provides a boost in performance over an unoptimised MPI-IO driver [11] – however, at scale PLFS overwhelms the Lustre file system used in this thesis. The results presented by Bent et al. show some performance numbers for both Lustre and GPFS, but report the largest gains on a PanFS installation, whereby specialised hardware and software is used to boost bandwidth. For most potential users of PLFS, it is likely more widespread file systems such as Lustre and GPFS will be in use and, as such, Chapter 5 outlined a method for rapidly evaluating the potential performance increase that PLFS could provide; the results in this thesis demonstrate an order of magnitude speed-up over the default system performance for some specific benchmarks and configurations.

The work in this thesis concluded in Chapter 6, with the comparison of the optimised Lustre driver (`ad_lustre`), which is often not available by default, to the unoptimised UNIX file system driver (`ad_ufs`). Previous work has demonstrated that better performance can be achieved with the `ad_lustre` driver and by exposing more of the parallelism inherent in the Lustre file system architecture [10, 76, 148]. Chapter 6 demonstrates similar results to previous studies, similarly showing that the best performance often comes from exploiting the maximum amount of parallelism available. This thesis has extended previous studies to demonstrate that the use of locally optimal settings may be harmful to global availability, where jobs that are competing for a shared resource (such as

a parallel file system) collide with each other, reducing respective performance. Further, this thesis has also evaluated the performance of PLFS at scale on a Lustre file system, showing that the PLFS architecture creates large amounts of self-contention, reducing the resources available to the entire system. The metrics derived in Chapter 6 will allow users of large shared systems to evaluate the impact their jobs may have on other users of the system, as well as allow procurement decisions to be made based on an expected I/O workload.

7.1 Limitations

This thesis concentrates largely on the write performance of particular benchmarks with little discussion of improving read performance. For HPC applications the initial state is usually loaded from an input deck, and from this point on the state is only written out to disk at particular intervals. Shan et al. suggest that write activities dominate on parallel machines because (i) post-processing and visualisation tasks are often performed on separate systems to the computation; (ii) most checkpoint files are never read back; and, (iii) input files are often very small [121]. Despite this, much of the work in this thesis is equally applicable to improving read performance. The work in Chapter 5 focuses on PLFS, a file system designed specifically to accelerate parallel write performance [11]. It has been shown, however, that PLFS also improves read performance [103], and much of the work on the Lustre file system in Chapter 6 will similarly improve both read and write performance.

Another potential limitation of this thesis is the use of unoptimised MPI-IO drivers in Chapters 4 and 5. However, the usage of the `ad_ufs` driver – or a lack of customised optimisation instructions – is common on large parallel systems, and is therefore representative of how these parallel machines are typically used. Unlike commodity clusters using GPFS file systems, there is a custom MPI-IO driver for the BlueGene architecture (`ad_bg1`) and its benefits were demonstrated in Chapter 4, where the BG/P achieved the highest average bandwidth

in most experiments. For the LLNL clusters a Lustre driver was built specifically for Chapter 6 and, again, demonstrated the benefit of optimised drivers for improving I/O throughput.

When performing experiments on a shared resource (such as a file system), the background machine load adds variability to the results. The machines used throughout this thesis were all production supercomputers at the time the experiments were performed, with a range of background loads. Minerva, being the smallest of the four systems, demonstrates the lowest variability of the clusters due to many background serial tasks that are not I/O-intensive. The two machines at LLNL use a shared file system, and due to the size and background load of these machines produce variable results. Finally, the decommissioned BG/P was not heavily loaded, but the architecture and use of aggregator nodes added variability to its performance. To allay these problems, full numerical data is given in the appendices, showing confidence intervals to capture the effect of background noise on the experiments performed.

One final limitation of this thesis is the small set of applications that were used throughout. All four benchmarks used are common throughout the literature and are also often used by industry to assess parallel file system performance. Two of the benchmarks are highly configurable applications that have been designed such that they can be made to replicate the I/O behaviour of other science applications and the final two applications recreate the performance characteristics of two production-grade codes. These four benchmarks are broadly representative of a large proportion of the I/O routines in many science codes; in particular two of the benchmarks perform their I/O through the HDF-5 library, which is widespread in science applications¹. The results in this thesis will therefore similarly apply to other applications and systems. Furthermore, much of the research in this thesis has been thoroughly tested against custom-written benchmarks designed to stress-test the results in order to demonstrate the applicability of the tools and techniques presented.

¹See <http://www.hdfgroup.org/users.html>

7.2 Future Work

The work in this thesis largely focuses on two themes: (i) the improvement of current generation I/O performance; and (ii) the procurement and potential performance of future I/O systems. Currently many users are experiencing poor performance from their systems due to a lack of optimisation and understanding of their applications I/O routines. Attempts have been made to bridge this gap with the use of auto-tuning to configure the MPI-IO options to get better performance [10], but this gives little regard to the system as a whole. As the results presented in this thesis demonstrate, a good quality of service can be achieved with parameter tuning. To reduce the burden on application developers, this parameter selection must be performed by the file system using some form of auto-tuning that is workload-aware. File systems such as Lustre must improve how resources are assigned, using some online monitoring and historical usage data to inform how resources should be distributed. While users continue to run their applications with “default” behaviours, the myth that current generation I/O performance is poor will remain. Optimisations made to file systems and applications to improve the current performance of storage systems will help inform future system developers by identifying some of the problems that will inevitably appear at exascale.

Procurement decisions for I/O systems currently rely on looking at the aggregate bandwidth provided by the OSSs and OSTs, and the capacity that is required. To make more sensible decisions about procurement, the usage patterns of the file system need to be better understood. One such study into doing this is being undertaken at the Argonne National Laboratory, using Darshan [21]. Currently the tools available either monitor an entire system with little information about the individual jobs, or monitor a specific job with little information about the rest of the system; by monitoring at both points, the current usage patterns and the users needs may be better understood.

Furthermore, the effects of background load upon I/O performance need

to be better understood. On capacity clusters (where there are many smaller jobs running simultaneously), the network and file system load can create large variances in runtimes, and this has long proved a problem in analytical performance modelling [59,98]. Understanding noisy environments will help to assess the performance potential of a file system better and aid the procurement process. This issue can be partially overcome using simulation, with some form of simulated machine noise (typically gaussian) to introduce the variances that are observed on production systems. However, the simulation platforms that exist at the time of writing [61,107] focus only on the compute performance at scale on shared-user machines; these must be extended to include I/O simulation, alongside background machine noise, to truly predict how applications will scale when they reach billion-way parallelism. Work has been done in simulating single disks and simple file systems [145] and with the addition of DiskSim, and a simulation platform like SST [107], the communication between compute nodes and I/O nodes can be investigated ahead of procurement, with the effect of OST placement being investigated to find smarter algorithms for data placement and resource allocation.

7.3 The Road to Exascale

Throughout this thesis, it has remained clear that current generation I/O systems will not scale up to exascale. With this in mind, either applications need to change how resilience is provided or storage systems require a redesign.

Many developers are reluctant to vastly change their stable codebases to change how fault tolerance is performed in their applications, therefore checkpointing will have to improve significantly, such that it does not become a bottleneck to performance at exascale [46]. Many of the results in this thesis offer small glimpses of what may be expected of I/O at exascale. Firstly the work in Chapter 4 shows how an extended I/O hierarchy may improve performance significantly [17]. On Daresbury's BG/P system data was aggregated by

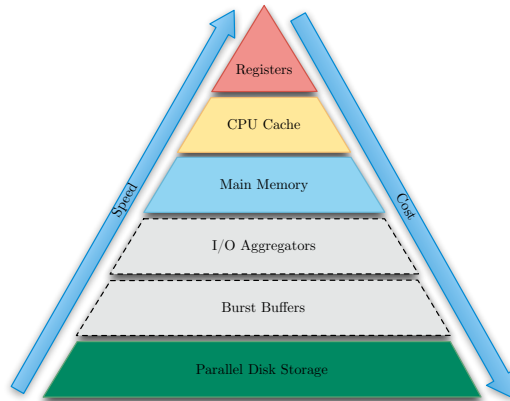


Figure 7.1: The memory hierarchy with potentially two additional layers for improved I/O performance on supercomputers.

dedicated high-performance nodes, after I/O operations had been performed on the compute nodes, and then sent to the file system. The file system also had a tiered architecture, where data was initially sent to fast fibre channel hard disk drives, before being committed to SATA disks. With the ever decreasing price, and increasing capacity, of solid state drives (SSDs), it seems likely that SSDs may be used as burst-buffers at exascale, helping to speed-match disk writes, before data is committed to spinning disks during computation [78, 96, 138]. Figure 7.1 shows how this augmented hierarchy may look.

The work that has been presented in Chapters 5 and 6 demonstrates one potential answer to the current I/O problem, showing that PLFS may benefit users running at small- to medium-scale, but does so by violating constraints in the file system (by exceeding the limit of Lustre stripes allowed). However, at large scale or on a large capacity system, PLFS either performs poorly or negatively impacts the rest of the file system for other users; at exascale, it is likely that without PLFS being re-engineered, this problem will get worse.

File systems will need to adapt to the upcoming challenges of billion-way parallelism by allowing more scalability and customisation. Load monitors may help the file system to exploit novel heuristics to make better decisions about where to place data blocks, reducing the risk of overloading particular storage

targets. The equations in Chapter 6 should allow for smarter procurement decisions to be made using an approximation of performance under load. Present-day large file system installations do perform much better than some literature would suggest, but only when used appropriately by the users. However, if users continue to run their applications with the default I/O configuration options, it is likely that I/O performance will continue to advance much slower than compute performance.

Recent efforts into reducing the time spent writing checkpoints have focussed on determining an optimal checkpointing period, such that the minimum amount of time is consumed by checkpointing activities to achieve a particular level of resilience. Cappello et al. investigate the use of preventive migration and preventive checkpointing, showing the preventive migration is better in the short term but that at large-scale (2^{20} nodes), both techniques will achieve poor utilisation unless the mean time between failures is large [18]. Aupy et al. have derived mathematic models to determine the optimum checkpointing period with respect to time and power consumption [5].

For developers willing to change the way their fault-tolerance is provided in their applications, the path to exascale may be clearer. While the MPI library contains mechanisms for handling application faults [57], some researchers have extended the MPI specification to provide more explicit fault-tolerance mechanisms [45]. However, even with MPI able to detect and continue in the presence of faults, techniques are required to recover lost computation.

Zheng et al. describe a double checkpointing algorithm, whereby each process is assigned a “buddy” process that they exchange an in-memory checkpoint with at regular intervals [153]. In the event of a process failure, the buddy process transfers a checkpoint to a new process which is swapped into the application to replace the failed process. Dongarra et al. extend this algorithm with a triple checkpointing algorithm that provides greater resilience at the expense of a small additional overhead [37].

Elliot et al. outline the use of partial redundancy in MPI applications, where

the communication routines in the MPI library are intercepted and the effort is replicated transparently to an application [43]. If a process in the system fails, their RedMPI library begins to redirect the traffic from a redundant process such that the application is unaware of the crashed process.

Regardless, present-day large file system installations do perform much better than some literature would suggest, but only when used appropriately by the users. However, if users continue to run their applications with the default I/O configuration options, it is likely that I/O performance will continue to advance much slower than compute performance.

Bibliography

- [1] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzeloni. Parallel I/O and the Metadata Wall. In *Proceedings of the 6th Workshop on Parallel Data Storage (PDSW'11)*, pages 13–18, Seattle, WA, 2011. ACM, New York, NY.
- [2] G. Alamási, C. Archer, J. G. Castaños, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, N. Smeds, B. Steinmacher-burrow, W. Gropp, and B. Toonen. Implementing MPI on the BlueGene/L Supercomputer. *Lecture Notes in Computer Science (LNCS)*, 3149:833–845, August–September 2004.
- [3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model – One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*, pages 95–105, Santa Barbara, CA, July 1995. ACM, New York, NY.
- [4] Argonne National Laboratory. Parallel I/O Benchmarking Consortium. <http://www.mcs.anl.gov/research/projects/pio-benchmark/> (accessed February 21, 2011), 2011.
- [5] G. Aupy, A. Benoit, T. Hérault, Y. Robert, and J. Dongarra. Optimal Checkpointing Period: Time vs. Energy. *Lecture Notes in Computer Science (LNCS)*, 8551:203–214, August 2014.
- [6] D. A. Bader, J. Berry, S. Kahan, R. Murphy, E. J. Riedy, and J. Willcock. Graph 500. <http://www.graph500.org> (accessed November 29, 2013), 2013.

- [7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, S. Fineberg, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. *The NAS Parallel Benchmarks*. NASA Ames Research Center, Moffet Field, CA, March 1994.
- [8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.
- [9] B. Behzad, S. Byna, S. M. Wild, and M. Snir. Improving Parallel I/O Autotuning with Performance Modeling. Technical Report ANL/MCS-P5066-0114, Argonne National Laboratory, Argonne, IL, January 2014.
- [10] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. Taming Parallel I/O Complexity with Autotuning. In *Proceedings of the 25th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, pages 1–12, Denver, CO, November 2013. IEEE Computer Society, Washington, DC.
- [11] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'09)*, pages 21:1–21:12, Portland, OR, November 2009. ACM, New York, NY.
- [12] R. F. Bird, S. J. Pennycook, S. A. Wright, and S. A. Jarvis. Towards a Portable and Future-proof Particle-in-Cell Plasma Physics Code. In

- Proceedings of the 1st International Workshop on OpenCL (IWOCL'13)*, Atlanta, GA, May 2013. Georgia Institute of Technology, GA.
- [13] R. F. Bird, S. A. Wright, D. A. Beckingsale, and S. A. Jarvis. Performance Modelling of Magnetohydrodynamics Codes. *Lecture Notes in Computer Science (LNCS)*, 7587:197–209, July 2013.
- [14] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The DiskSim Simulation Environment Version 4.0 Reference Manual, 2008.
- [15] Bull. *BullX Cluster Suite Application Developer's Guide*. Les Clayes-sous-Bois, Paris, April 2010.
- [16] R. A. Bunt, S. J. Pennycook, S. A. Jarvis, B. L. Lapworth, and Y. K. Ho. Model-Led Optimisation of a Geometric Multigrid Application. In *Proceedings of the 15th IEEE International Conference on High Performance Computing and Communications (HPCC'13)*, pages 1–12. IEEE Computer Society, Los Alamitos, CA, November 2013.
- [17] D. Camp, H. Childs, A. Chourasia, C. Garth, and K. I. Joy. Evaluating the Benefits of an Extended Memory Hierarchy for Parallel Streamline Algorithms. In *The 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV'11)*, pages 57–64, Providence, RI, October 2011. IEEE Computer Society, Los Alamitos, CA.
- [18] F. Cappello, H. Casanova, and Y. Robert. Preventive Migration vs. Preventive Checkpointing for Extreme Scale Supercomputers. *Parallel Processing Letters*, 21(2):111–132, June 2011.
- [19] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, November 2009.
- [20] R. Card, T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. In *The Proceedings of the 1st Dutch Inter-*

- national Symposium on Linux*, Amsterdam, The Netherlands, December 1994. State University of Groningen, The Netherlands.
- [21] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 Characterization of Petascale I/O Workloads. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09)*, pages 1–10, New Orleans, LA, September 2009. IEEE Computer Society, Los Alamitos, CA.
- [22] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference (ALS'00)*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.
- [23] P. H. Carns, B. W. Settlemyer, and W. B. Ligon III. Using Server-to-Server Communication in Parallel File Systems to Simplify Consistency and Improve Performance. In *Proceedings of the 20th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*, pages 6:1–6:8, Austin, TX, 2008. IEEE Computer Society Press, Piscataway, NJ.
- [24] Y. Chanel. Study of the Lustre File System Performances Before its Installation in a Computing Cluster. <http://upcommons.upc.edu/pfc/bitstream/2099.1/5626/1/49964.pdf> (accessed June 20, 2011), 2008.
- [25] Y. Chen. Towards Scalable I/O Architecture for Exascale Systems. In *Proceedings of the 2011 ACM International Workshop on Many Task Computing on Grids and Supercomputers (MTAGS'11)*, pages 43–48, Seattle, WA, November 2011. ACM New York, NY.
- [26] Y. Chen, Y. Lu, P. Amritkar, R. Thakur, and Y. Zhuang. Performance model-directed data sieving for high-performance i/o. *The Journal of Supercomputing*, pages 1–25, September 2014.

- [27] I. D. Chivers and J. Sleightholme. An Introduction to Sun Studio. *ACM SIGPLAN Fortran Forum*, 28(1):13–25, April 2009.
- [28] W. chun Feng and K. W. Cameron. The Green 500. <http://www.green500.org> (accessed February 20, 2014), 2014.
- [29] J. Cope, M. Oberg, H. M. Tufo, and M. Woitaszek. Shared Parallel Filesystems in Heterogeneous Linux Multi-Cluster Environments. In *Proceedings of the 6th LCI International Conference on Linux Clusters*, pages 1–21, Chapel Hill, NC, 2005. Linux Clusters Institute, NM.
- [30] D. Culler, R. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’93)*, pages 1–12, San Diego, CA, July 1993. ACM, New York, NY.
- [31] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January–March 1998.
- [32] J. A. Davis, G. R. Mudalige, S. D. Hammond, J. A. Herdman, I. Miller, and S. A. Jarvis. Predictive Analysis of a Hydrodynamics Application on Large-Scale CMP Clusters. *Computer Science – Research and Development*, 26(3–4):175–185, June 2011.
- [33] Department of Energy. Department of Energy Exascale Strategy – Report to Congress. Technical report, United States Department of Energy, Washington, DC 20585, June 2013.
- [34] V. Deshpande, X. Wu, and F. Mueller. Auto-Generation of Communication Benchmark Traces. *SIGMETRICS Performance Evaluation Review*, 40(2):99–105, October 2012.

- [35] P. M. Dickens and J. Logan. Towards a High Performance Implementation of MPI-IO on the Lustre File System. In *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems (OTM'08)*, pages 870–885, Monterrey, Mexico, January 2008. Springer-Verlag, Berlin, Heidelberg.
- [36] P. M. Dickens and J. Logan. A High Performance Implementation of MPI-IO for a Lustre File System Environment. *Concurrency and Computation: Practice & Experience*, 22(11):1433–1449, August 2010.
- [37] J. Dongarra, T. Hérault, and Y. Robert. Performance and Reliability Trade-offs for the Double Checkpointing Algorithm. *International Journal of Networking and Computing*, 4(1):23–41, 2014.
- [38] J. J. Dongarra. The MPI Profiling Interface. <http://www.netlib.org/utk/papers/mpi-book/node182.html> (accessed January 29, 2013), 1995.
- [39] J. J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, University of Tennessee, Knoxville, TN, February 2013.
- [40] S. S. Dosanjh, R. F. Barrett, D. W. Doerfler, S. D. Hammond, K. S. Hemmert, M. A. Heroux, P. T. Lin, K. T. Pedretti, A. F. Rodrigues, T. G. Trucano, and J. P. Luitjens. Exascale Design Space Exploration and Co-design. *Future Generation Computer Systems*, 30:46–58, January 2014.
- [41] O. Drokin. Lustre File Striping Across a Large Number of OSTs. In *Proceedings of the 2011 Lustre User Group (LUG'13)*, pages 1–17, Orlando, FL, April 2011. OpenSFS, LUG.
- [42] P. R. Effert and D. S. Parker. File Systems in User Space. In *Proceedings of the 1993 USENIX Winter Conference*, pages 229–240, San Diego, CA, January 1993. USENIX Association, Berkeley, CA.

- [43] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelman. Combining Partial Redundancy and Checkpointing for HPC. In *Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems (ICDCS'12)*, pages 615–626, Macau, China, June 2012. IEEE Computer Society, Washington, DC.
- [44] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.
- [45] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. *Lecture Notes in Computer Science (LNCS)*, 1908:346–353, 2000.
- [46] D. Fiala, F. Mueller, C. Engelman, R. Riesen, K. Ferreira, and R. Brightwell. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In *Proceedings of the 24th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, pages 78:1–78:12, Salt Lake City, UT, November 2012. IEEE Computer Society, Washington, DC.
- [47] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *The Astrophysical Journal Supplement Series*, 131(1):273, 2000.
- [48] K. Fuerlinger, N. J. Wright, and D. Skinner. Effective Performance Measurement at Petascale Using IPM. In *Proceedings of the IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS'10)*, pages 373–380, Shanghai, China, December 2010. IEEE Computer Society, Washington, DC.
- [49] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Cas-

- tain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. *Lecture Notes in Computer Science (LNCS)*, 3241:97–104, September 2004.
- [50] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice & Experience*, 22(6):702–719, April 2010.
- [51] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly. Performance Analysis of the OP2 Framework on Many-Core Architectures. *SIGMETRICS Performance Evaluation Review*, 38(4):9–15, March 2011.
- [52] J. Gim and Y. Won. Extract and Infer Quickly: Obtaining Sector Geometry of Modern Hard Disk Drives. *ACM Transactions on Storage (TOS)*, 6(2):6–26, July 2010.
- [53] J. Gim and Y. Won. Relieving the Burden of Track Switch in Modern Hard Disk Drives. *Multimedia Systems*, 17(3):219–235, June 2011.
- [54] J. Gim, Y. Won, J. Chang, J. Shim, and Y. Park. DIG: Rapid Characterization of Modern Hard Disk Drive and Its Performance Implication. In *Proceedings of the 5th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI’08)*, pages 74–83. IEEE Computer Society, Washington, DC, September 2008.
- [55] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction (SIGPLAN’82)*, pages 120–126. ACM New York, NY, April 1982.
- [56] W. Gropp. MPICH2: A New Start for MPI Implementations. *Lecture Notes in Computer Science (LNCS)*, 2474:7, October 2002.

- [57] W. Gropp and E. Lusk. Fault Tolerance in Message Passing Interface Programs. *International Journal of High Performance Computing Applications*, 18(3):363–372, August 2004.
- [58] T. J. Hacker, F. Romero, and C. D. Carothers. An Analysis of Clustered Failures on Large Supercomputing Systems. *Journal of Parallel and Distributed Computing (JPDC)*, 69(7):652–665, July 2009.
- [59] S. D. Hammond. *Performance Modelling and Simulation of High Performance Computing Systems*. PhD thesis, University of Warwick, Coventry, UK, 2011.
- [60] S. D. Hammond, G. R. Mudalige, J. A. Smith, J. A. Davis, A. B. Mills, S. A. Jarvis, J. Holt, I. Miller, J. A. Herdman, and A. Vadgama. Performance Prediction and Procurement in Practice: Assessing the Suitability of Commodity Cluster Components for Wavefront Codes. *IET Software*, 3(6):509–521, December 2009.
- [61] S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, J. A. Herdman, and A. Vadgama. WARPP: A Toolkit for Simulating High-Performance Parallel Scientific Codes. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques (Simutools’09)*, pages 19:1–19:10, Rome, Italy, 2009. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, Brussels, Belgium.
- [62] J. He, J. Bent, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun. Discovering Structure in Unstructured I/O. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC’12)*, pages 1–6, Salt Lake City, UT, November 2012. IEEE Computer Society, Washington, DC.
- [63] R. Hedges, K. Fitzgerald, M. Gary, and D. M. Stearman. Comparison of Leading Parallel NAS File Systems on Commodity Hardware. In *Proceed-*

- ings of the 5th Annual Workshop on Petascale Data Storage (PDSW'10)*, page 1, New Orleans, LA, November 2010. IEEE Computer Society, Los Alamitos, CA.
- [64] T. Hey. Richard Feynman and Computation. *Contemporary Physics*, 40(4):257–265, 1999.
- [65] C. Hill, C. DeLuca, V. Balaji, M. Suarez, and A. da Silva. The Architecture of the Earth System Modeling Framework. *Computing in Science and Engineering*, 6(1):18–28, January 2004.
- [66] M. Howison. Tuning HDF5 for Lustre File Systems. In *Proceedings of the Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS'10)*, Heraklion, Crete, Greece, September 2012. IEEE Computer Society, Los Alamitos, CA.
- [67] W. Hsu and A. J. Smith. The Performance Impact of I/O Optimizations and Disk Improvements. *IBM Journal of Research and Development*, 48(2):255–289, March 2004.
- [68] L. P. Huse, K. Omang, H. O. Bugge, H. Ry, A. T. Haugsdal, and E. Rustad. ScaMPI – Design and Implementation. *Lecture Notes in Computer Science (LNCS)*, 1734:249–261, January 1999.
- [69] Intel. Intel VTune Amplifier XE 2013. <https://software.intel.com/en-us/intel-vtune-amplifier-xe> (accessed December 20, 2013), 2013.
- [70] S. Ishiguro, J. Murakami, Y. Oyama, and O. Tatebe. Optimizing Local File Accesses for FUSE-Based Distributed Storage. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC'12)*, pages 760–765, Salt Lake City, UT, November 2012. IEEE Computer Society, Washington, DC.
- [71] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. A Performance Comparison Between the Earth Simulator and Other Terascale Systems on

- a Characteristic ASCI Workload: Research Articles. *Concurrency and Computation: Practice & Experience*, 17(10):1219–1238, August 2005.
- [72] D. E. Knuth. Ancient Babylonian Algorithms. *Communications of the ACM Magazine*, 15(7):671–677, July 1972.
- [73] Q. Koziol and R. Matzke. *HDF5 – A New Generation of HDF: Reference Manual and User Guide*. Champaign, IL, 1998.
- [74] J. Layton. HPC Storage – Getting Started with I/O Profiling. <http://hpc.admin-magazine.com/Articles/HPC-Storage-I-O-Profiling> (accessed February 02, 2012), 2012.
- [75] J. Li, W. keng Liao, A. N. Choudhary, R. B. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the 15th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC’03)*, pages 39–50, Pheonix, AZ, November 2003. ACM, New York, NY.
- [76] B. Lind. Lustre Tuning Parameters. In *Proceedings of the 2013 Lustre User Group (LUG’13)*, pages 1–12, San Diego, CA, April 2013. OpenSFS, LUG.
- [77] N. Liu, C. D. Carothers, J. Cope, P. H. Carns, R. B. Ross, A. Crume, and C. Maltzahn. Modeling a Leadership-Scale Storage System. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics (PPAM’11)*, pages 10–19, Torun, Poland, 2012. Springer-Verlag, Berlin, Heidelberg.
- [78] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn. On the Role of Burst Buffers in Leadership-Class Storage Systems. In *Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST’12)*, pages 1–11, Pacific Grove, CA, April 2012. IEEE Computer Society, Los Alamitos, CA.

- [79] J. Logan and P. M. Dickens. Towards an Understanding of the Performance of MPI-IO in Lustre File Systems. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing (CLUSTER'08)*, pages 330–335, Tsukuba, Ibaraki, Japan, September 2008. IEEE Computer Society, Los Alamitos, CA.
- [80] Los Alamos National Laboratory. PLFS Manual. <http://institute.lanl.gov/plfs/man/index.html> (accessed June 1, 2014), 2014.
- [81] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 18th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'06)*, pages 213:1–213:105, Tampa, FL, November 2006. ACM New York, NY.
- [82] M. Massie. Ganglia Monitoring System. <http://ganglia.sourceforge.net> (accessed January 16, 2012), 2011.
- [83] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. <http://www.cs.virginia.edu/stream/> (accessed May 12, 2014), 1995.
- [84] K. Mehta, J. Bent, A. Torres, G. Grider, and E. Gabriel. A Plugin for HDF5 Using PLFS for Improved I/O Performance and Semantic Analysis. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC'12)*, pages 746–752, Salt Lake City, UT, November 2012. IEEE Computer Society, Washington, DC.
- [85] M. Mesnier, G. R. Ganger, and E. Riedel. Object-Based Storage. *IEEE Communications Magazine*, 41(8):84–90, August 2003.
- [86] Message Passing Interface Forum. MPI: A Message Passing Interface Standard Version 2.2. *High Performance Computing Applications*, 12(1–2):1–647, 2009.

- [87] H. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon. Top 500 Supercomputer Sites. <http://top500.org> (accessed May 5, 2013), 2013.
- [88] E. Molina-Estolano, C. Maltzahn, J. Bent, and S. A. Brandt. Building a Parallel File System Simulator. *Journal of Physics: Conference Series*, 180(1), 2009.
- [89] G. R. Mudalige, S. D. Hammond, J. A. Smith, and S. A. Jarvis. Predictive Analysis and Optimisation of Pipelined Wavefront Computations. In *Proceedings of the 23rd IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, pages 1–8, Rome, Italy, May 2009. IEEE Computer Society, Washington, DC.
- [90] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, January 1996.
- [91] D. F. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage. In *Proceedings of the 16th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'04)*, pages 53–62, Washington, DC, USA, 2004. IEEE Computer Society.
- [92] B. Nitzberg and V. M. Lo. Collective Buffering: Improving Parallel I/O Performance. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC'97)*, pages 148–157, Portland, OR, August 1997. IEEE Computer Society, Washington, DC.
- [93] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. ScalaTrace: Scalable Compression and Replay of Communication Traces for High-Performance Computing. *Journal of Parallel and Distributed Computing (JPDC)*, 69(8):696–710, August 2009.
- [94] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield. Zest: Checkpoint Storage System for Large Supercomputers. In *Proceedings of the*

- 3rd Annual Workshop on Petascale Data Storage (PDSW'08)*, pages 1–5, Austin, TX, November 2008. IEEE Computer Society, Los Alamitos, CA.
- [95] J. Nunez and J. Bent. MPI-IO Test User’s Guide. <http://institutes.lanl.gov/data/software/> (accessed February 21, 2011), 2011.
- [96] X. Ouyang, S. Marcarelli, and D. Panda. Enhancing Checkpoint Performance with Staging IO and SSD. In *Proceedings of the 8th IEEE International Workshop on Storage Network Architecture and Parallel I/O (SNAPI'10)*, pages 13–20, Incline Village, NV, May 2010. IEEE Computer Society, Los Alamitos, CA.
- [97] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*, pages 109–116, Chicago, IL, 1988. ACM, New York, NY.
- [98] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Wright, and S. A. Jarvis. On the Acceleration of Wavefront Applications using Distributed Many-Core Architectures. *The Computer Journal*, 55(2):138–153, February 2012.
- [99] S. J. Pennycook, S. D. Hammond, S. A. Wright, J. A. Herdman, I. Miller, and S. A. Jarvis. An Investigation of the Performance Portability of OpenCL. *Journal of Parallel and Distributed Computing (JPDC)*, 73(11):1439–1450, November 2013.
- [100] J. Piernas Cánovas and J. Nieplocha. Implementation and Evaluation of Active Storage in Modern Parallel File Systems. *Parallel Computing*, 36(1):26–47, January 2010.
- [101] J. Piernas Cánovas, J. Nieplocha, and E. J. Felix. Evaluation of Active Storage Strategies for the Lustre Parallel File System. In *Proceedings of the 19th ACM/IEEE International Conference for High Performance*

- Computing, Networking, Storage and Analysis (SC'07)*, pages 1–10, Reno, NV, November 2007. ACM New York, NY.
- [102] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, March 1995.
- [103] M. Polte, J. F. Lofstead, J. Bent, G. Gibson, S. A. Klasky, Q. Liu, M. Parashar, K. Schwan, and M. Wolf. ... And Eat It Too: High Read Performance in Write-Optimized HPC I/O Middleware File Formats. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage (PDSW'09)*, pages 21–25, Portland, OR, November 2009. ACM, New York, NY.
- [104] M. Polte, J. Simsa, W. Tantisiriroj, G. Gibson, S. Dayal, M. Chainani, and D. K. Uppugandla. Fast Log-based Concurrent Writing of Checkpoints. In *Proceedings of the 3rd Annual Workshop on Petascale Data Storage (PDSW'08)*, pages 1–4, Austin, TX, November 2008. IEEE Computer Society, Los Alamitos, CA.
- [105] I. Raicu, I. T. Foster, and P. Beckman. Making a Case for Distributed File Systems at Exascale. In *Proceedings of the 3rd International Workshop on Large-scale System and Application Performance (LSAP'11)*, pages 11–18. ACM, New York, NY, June 2011.
- [106] R. K. Rew and G. P. Davis. NetCDF: An Interface for Scientific Data Access. *IEEE Computer Graphics and Applications*, 10(4):76–82, 1990.
- [107] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. I. Weston, R. Riesen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. The Structural Simulation Toolkit. *SIGMETRICS Performance Evaluation Review*, 38(4):37–42, March 2011.
- [108] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

- [109] R. Rosner, A. Calder, J. Dursi, B. Fryxell, D. Q. Lamb, J. C. Niemeyer, K. Olson, P. Ricker, F. X. Timmes, J. W. Truran, H. Tufo, Y. nan Young, and M. Zingale. Flash Code: Studying Astrophysical Thermonuclear Flashes. *Computing in Science & Engineering*, 2(2):33–41, March–April 2000.
- [110] M. S. Rothberg. Disk Drive Refreshing Zones Based On Serpentine Access of Disk Surfaces. US Patent 7872822 <http://www.google.com/patents/US7872822>, January 2011.
- [111] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *Computer*, 27(3):17–28, March 1994.
- [112] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *The Proceedings of the USENIX 1985 Summer Conference*, pages 119–130, Portland, OR, June 1985. USENIX Association.
- [113] J. Schindler and G. R. Ganger. Automated Disk Drive Characterization. Technical Report CMU-CS-99-176, Parallel Data Laboratory, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1999.
- [114] J. Schindler and G. R. Ganger. Automated Disk Drive Characterization. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'00)*, pages 112–113, Santa Clara, CA, 2000. ACM, New York, NY.
- [115] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association Berkeley, CA.

- [116] B. Schroeder and G. A. Gibson. Understanding Failures in Petascale Computers. *Journal of Physics: Conference Series*, 78(1):012022, July 2007.
- [117] P. Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proceedings of the Linux Symposium*, pages 380–386, Ottawa, Ontario, Canada, July 2003. The Linux Symposium.
- [118] M. Seger. Collectl. <http://collectl.sourceforge.net> (accessed January 16, 2012), 2011.
- [119] J. Shalf, S. S. Dosanjh, and J. Morrison. Exascale Computing Technology Challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR’10)*, pages 1–25. Springer-Verlag Berlin, Heidelberg, January 2011.
- [120] H. Shan, K. Antypas, and J. Shalf. Characterizing and Predicting the I/O Performance of HPC Applications using a Parameterized Synthetic Benchmark. In *Proceedings of the 20th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC’08)*, Austin, TX, November 2008. IEEE Press Piscataway, NJ.
- [121] H. Shan and J. Shalf. *Using IOR to Analyze the I/O Performance for HPC Platforms*. Lawrence Berkeley National Laboratory, Berkeley, CA, May 2007.
- [122] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.
- [123] E. Shriver, A. Merchant, and J. Wilkes. An Analytic Behavior Model for Disk Drives with Readahead Caches and Request Reordering. *SIGMETRICS Performance Evaluation Review*, 26(1):182–191, June 1998.

- [124] G. E. Suh, S. Devadas, and L. Rudolph. Analytical Cache Models with Applications to Cache Partitioning. In *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*, pages 1–12, Sorrento, Italy, 2001. ACM, New York, NY.
- [125] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS'96)*, pages 180–187, Annapolis, MD, October 1996. IEEE Computer Society, Los Alamitos, CA.
- [126] R. Thakur, W. Gropp, and E. Lusk. Data-Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS'99)*, pages 182–191, Annapolis, MD, February 1999. IEEE Computer Society, Los Alamitos, CA.
- [127] R. Thakur, E. Lusk, and W. Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, October 1997.
- [128] The Mantevo Project. The Mantevo Benchmarks. <https://mantevo.org/> (accessed November 1, 2014), 2014.
- [129] C. A. Thekkath, J. Wilkes, and E. D. Lazowska. Techniques for File System Simulation. *Software: Practice and Experience*, 24(11):981–999, 1994.
- [130] D. Thiébaud, H. S. Stone, and J. L. Wolf. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Transactions on Computers*, 41(6):665–676, June 1992.
- [131] W. F. Tichy and Z. Ruan. Towards a Distributed File System. Technical Report CSD-TR-480, Perdue University, West Lafayette, IL, May 1984.

- [132] D. Toussaint, R. van de Water, R. Sugar, U. Heller, S. Gottlieb, M. Oktay, J. Foley, J. Laiho, C. Winterowd, A. Bazavov, M. Lightman, J. Kim, L. Levkova, C. Bernard, C. DeTar, R. Zhou, S. Qiu, S. Lee, J. Osborn, and J. Hetrick. The MIMD Latice Computation (MILC) Collaboration. <http://www.physics.utah.edu/~detar/milc/> (accessed November 1, 2014), 2014.
- [133] S. Tweedie. Journalling the ext2fs File System. In *Proceedings of the Linux Expo 1998 (LinuxExpo'98)*, Durham, NC, May 1998. Duke University, Durham, NC.
- [134] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. Scalable I/O tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage (PDSW'09)*, pages 26–31, Portland, OR, November 2009. ACM, New York, NY.
- [135] Y. Wang and D. Kaeli. Profile-Guided I/O Partitioning. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03)*, pages 252–260, San Francisco, CA, June 2003. ACM, New York, NY.
- [136] Y. Wang and D. Kaeli. Source Level Transformations to Improve I/O Data Partitioning. In *Proceedings of the 1st International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'03)*, pages 27–35, New Orleans, LA, September 2003. ACM, New York, NY.
- [137] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 307–320, Seattle, WA, November 2006. USENIX Association.
- [138] B. Welch and G. Noer. Optimizing a Hybrid SSD/HDD HPC Storage System Based on File Size Distributions. In *Proceedings of the 29th IEEE Symposium on Mass Storage Systems and Technologies (MSST'13)*, pages

- 1–12, Long Beach, CA, May 2013. IEEE Computer Society, Washington, DC.
- [139] B. L. Wolman and T. M. Olson. IOBENCH: A System Independent IO Benchmark. *ACM SIGARCH Computer Architecture News*, 17(5):55–70, 1989.
- [140] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. *SIGMETRICS Performance Evaluation Review*, 22(1):241–251, May 1994.
- [141] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. H. Bhalerao, and S. A. Jarvis. Parallel File System Analysis Through Application I/O Tracing. *The Computer Journal*, 56(2):141–155, February 2013.
- [142] S. A. Wright, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. Lightweight Parallel I/O Analysis at Scale. *Lecture Notes in Computer Science (LNCS)*, 6977:235–249, October 2011.
- [143] S. A. Wright, S. D. Hammond, S. J. Pennycook, I. Miller, J. A. Herdman, and S. A. Jarvis. LDPLFS: Improving I/O Performance without Application Modification. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium Workshops & PhD Forum (IPDPSW’12)*, pages 1352–1359, Shanghai, China, 2012. IEEE Computer Society, Washington, DC.
- [144] S. A. Wright, S. J. Pennycook, S. D. Hammond, and S. A. Jarvis. RIOT – A Parallel Input/Output Tracer. In *Proceedings of the 27th Annual UK Performance Engineering Workshop (UKPEW’11)*, pages 25–39, Bradford, UK, July 2011. The University of Bradford, Bradford, UK.
- [145] S. A. Wright, S. J. Pennycook, and S. A. Jarvis. Towards the Automated Generation of Hard Disk Models Through Physical Geometry Discovery.

- In *Proceedings of the 3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS'12)*, pages 1–8, Salt Lake City, UT, November 2012. ACM, New York, NY.
- [146] M. Yang and Q. Koziol. Using Collective IO Inside a High Performance IO Software Package – HDF5. Technical report, National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign, IL, 2006.
- [147] C. S. Yoo, R. Sankaran, and J. H. Chen. Three-dimensional Direct Numerical Simulation of a Turbulent Lifted Hydrogen Jet Flame in Heated Coflow: Flame Stabilization and Structure. *Journal of Fluid Mechanics*, 640:453–481, December 2009.
- [148] H. You, Q. Liu, Z. Li, and S. Moore. The Design of an Auto-Tuning I/O Framework on Cray XT5 System. In *Proceedings of the 2011 Cray Users Group Conference (CUG'11)*, pages 1–10, Fairbanks, AK, May 2011. Cray Inc.
- [149] Y. Yuan, Y. Wu, Q. Wang, G. Yang, and W. Zheng. Job Failures in High Performance Computing Systems: A Large-scale Empirical Study. *Computers & Mathematics with Applications*, 63(2):365–377, January 2012.
- [150] S. Zangenehpour. Method and Apparatus for Positioning Head of Disk Drive Using Zone-Bit-Recording. US Patent 5257143 <http://www.google.com/patents/US5257143>, October 1993.
- [151] S. Zertal. Using Solid State Disks for HPC Applications. In *Proceedings of the International Conference on Computing, Networking and Digital Technologies (ICCNDT'12)*, pages 209–217, Sanad, Bahrain, 2012. The Society of Digital Information and Wireless Communication.
- [152] T. Zhao, V. March, S. Dong, and S. See. Evaluation of a Performance Model of Lustre File System. In *Proceedings of the 5th Annual Chi-*

- naGrid Conference (ChinaGrid'10)*, pages 191–196, Guangzhou, Guangdong, China, 2010. IEEE Computer Society, Washington, DC.
- [153] G. Zheng, L. Shi, and L. V. Kale. FTC-Charm++: An In-memory Checkpoint-based Fault Tolerant Runtime for Charm++ and MPI. In *Proceedings of the 2004 IEEE Conference on Cluster Computing (CLUSTER'04)*, pages 93–103, San Diego, CA, September 2004. IEEE Computer Society, Los Alamitos, CA.
- [154] Y. Zhu and Y. Hu. Disk Built-in Caches: Evaluation on System Performance. In *The 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS'03)*, pages 306–313, Orlando, FL, October 2003. IEEE Computer Society, Los Alamitos, CA.
- [155] M. Zingale. FLASH I/O Benchmark Routine – Parallel HDF 5. http://www.ucolick.org/~zingale/flash_benchmark_io/ (accessed February 21, 2011), 2011.

APPENDIX A

RIOT Feasibility Study – Additional Results

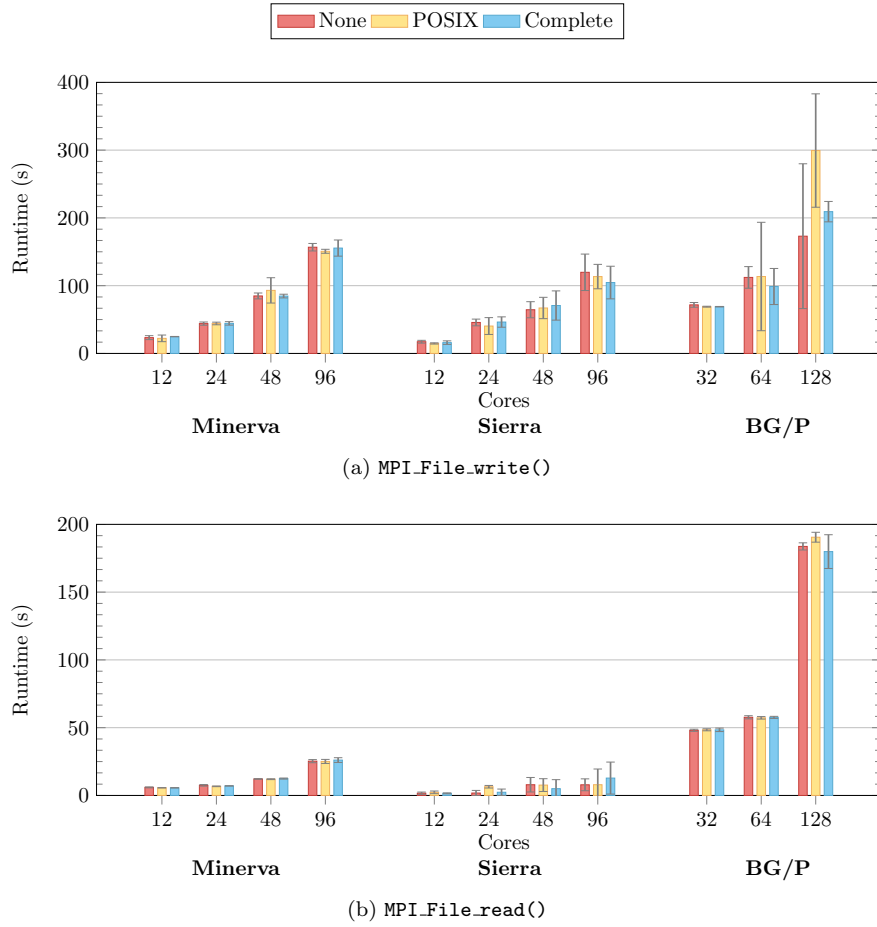


Figure A.1: Total runtime of RIOT overhead analysis software for the functions `MPI_File_write()` and `MPI_File_read()`, on three platforms, with three different configurations: No RIOT tracing, POSIX RIOT tracing and complete RIOT tracing.

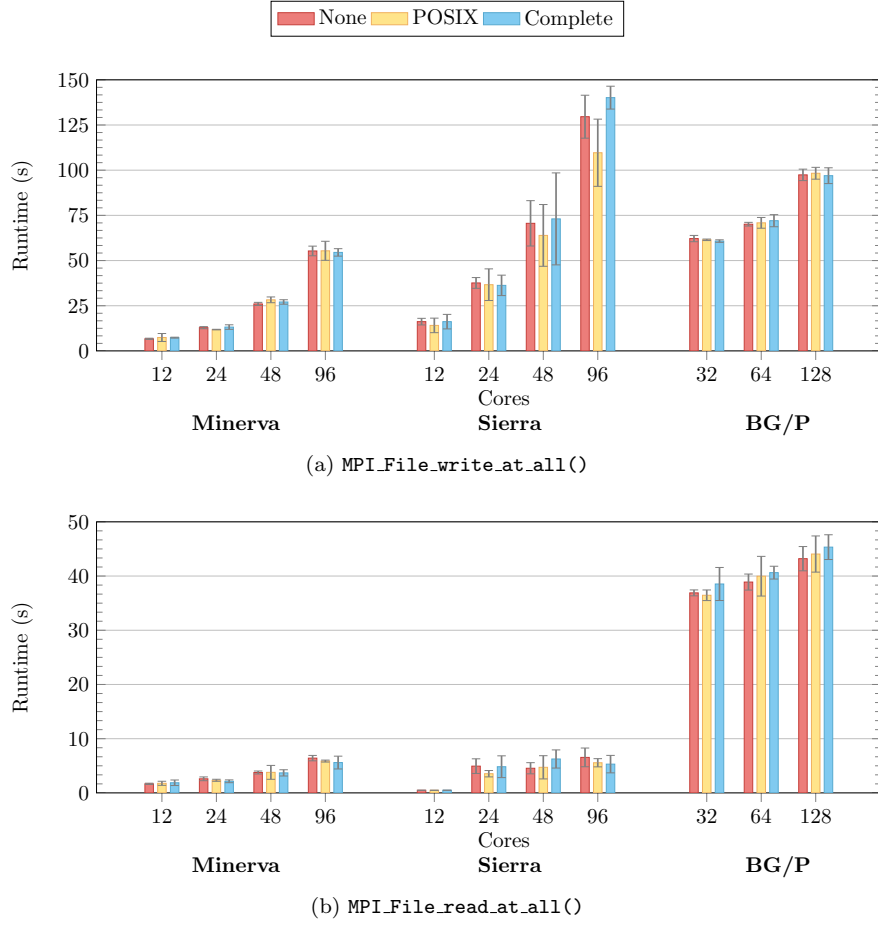


Figure A.2: Total runtime of RIOT overhead analysis software for the functions `MPI_File_write_at_all()` and `MPI_File_read_at_all()`, on three platforms, with three different configurations: No RIOT tracing, POSIX RIOT tracing and complete RIOT tracing.

	AWE	IOR	NPB	S3D	Mantevo	HDF5	MILC	LAMMPS	ESMF	Total
	[120]	[7]	[147]		[128]	[73]	[132]	[102]	[65]	
read	15	1	0	0	0	2	2	0	0	21
read_all	43	1	0	12	44	0	0	3	4	107
read_at	13	1	1	0	0	17	0	1	1	34
read_at_all	41	1	1	0	0	7	0	2	0	53
read_ordered	47	1	0	0	0	0	0	0	0	48
read_shared	15	0	0	0	0	0	0	0	0	15
write	27	1	0	0	12	3	2	0	0	45
write_all	35	1	0	12	12	0	0	3	3	66
write_at	11	1	1	0	0	19	0	9	4	45
write_at_all	35	1	1	0	0	8	0	7	0	54
write_ordered	50	1	0	0	0	0	0	0	0	51
write_shared	15	0	0	0	0	0	0	0	0	15

Table A.1: Incidence of `MPI_File_*` function calls in 9 application suites, benchmarks and I/O libraries.

Tracing	Minerva			Sierra			BG/P		
	24	48	96	24	48	96	32	64	128
MPIFile.write()									
None	44.00	84.84	156.75	45.71	64.49	119.72	71.68	112.17	173.02
POSIX	44.22	93.05	150.72	40.44	67.07	113.41	68.99	113.52	299.38
All	44.36	84.66	155.48	46.33	70.72	104.61	69.00	98.76	209.22
<i>Change (%)</i>	<i>0.82</i>	<i>0.21</i>	<i>0.81</i>	<i>1.35</i>	<i>9.67</i>	<i>12.63</i>	<i>3.74</i>	<i>11.95</i>	<i>20.92</i>
MPIFile.write.all()									
None	26.65	51.64	101.61	39.58	76.12	124.45	71.37	100.51	137.85
POSIX	26.17	51.95	101.08	38.59	68.44	131.63	70.59	100.93	135.17
All	26.99	50.66	99.95	38.11	64.32	127.30	70.70	100.09	139.02
<i>Change (%)</i>	<i>1.28</i>	<i>1.92</i>	<i>1.64</i>	<i>3.70</i>	<i>15.50</i>	<i>2.29</i>	<i>0.95</i>	<i>0.42</i>	<i>0.85</i>
MPIFile.write.at.all()									
None	12.86	26.00	55.31	37.61	70.60	129.59	62.16	70.12	97.41
POSIX	11.81	28.27	55.39	36.64	63.91	109.65	61.48	70.87	98.30
All	13.20	27.08	54.51	36.27	73.06	140.14	60.78	72.06	96.96
<i>Change (%)</i>	<i>2.59</i>	<i>4.16</i>	<i>1.44</i>	<i>3.57</i>	<i>3.48</i>	<i>8.14</i>	<i>2.22</i>	<i>2.77</i>	<i>0.46</i>
MPIFile.read()									
None	7.46	12.17	25.30	1.79	7.91	7.87	48.09	57.67	183.73
POSIX	6.73	11.99	25.10	5.43	7.64	7.92	48.45	57.27	190.51
All	7.05	12.46	26.10	2.36	5.00	12.82	48.41	57.60	179.90
<i>Change (%)</i>	<i>5.52</i>	<i>2.37</i>	<i>3.16</i>	<i>32.11</i>	<i>36.78</i>	<i>62.98</i>	<i>0.67</i>	<i>0.13</i>	<i>2.08</i>
MPIFile.read.all()									
None	21.64	26.29	41.45	15.67	28.06	53.38	57.86	68.32	205.37
POSIX	19.85	27.17	42.31	18.63	24.73	58.97	56.27	64.52	211.50
All	20.67	26.62	41.24	16.99	25.74	63.82	58.97	68.57	209.91
<i>Change (%)</i>	<i>4.50</i>	<i>1.22</i>	<i>0.51</i>	<i>8.43</i>	<i>8.29</i>	<i>19.56</i>	<i>1.91</i>	<i>0.37</i>	<i>2.21</i>
MPIFile.read.at.all()									
None	2.60	3.76	6.42	4.94	4.55	6.55	36.89	38.89	43.20
POSIX	2.33	3.79	5.87	3.54	4.72	5.56	36.45	39.97	44.06
All	2.14	3.70	5.60	4.83	6.26	5.31	38.53	40.63	45.33
<i>Change (%)</i>	<i>17.58</i>	<i>1.78</i>	<i>12.74</i>	<i>2.21</i>	<i>37.58</i>	<i>18.96</i>	<i>4.44</i>	<i>4.47</i>	<i>4.93</i>

Table A.2: Average time (s) to perform one hundred 4 MB operations: without RIOT, with only POSIX tracing and with complete MPI and POSIX RIOT tracing. The change in time is shown between full RIOT tracing and no RIOT tracing.

APPENDIX B

Numeric Data for Perceived and Effective Bandwidth

Cores	Perceived MPI		Effective MPI		Effective POSIX	
	B/W	95% CI	B/W	95% CI	B/W	95% CI
Minerva						
12	34.920	(26.942, 42.899)	2.907	(2.243, 3.571)	11.105	(8.640, 13.571)
24	44.742	(41.636, 47.848)	1.806	(1.702, 1.911)	4.916	(4.234, 5.597)
48	57.940	(51.178, 64.702)	1.064	(0.993, 1.134)	2.335	(1.737, 2.934)
96	65.293	(58.449, 72.137)	0.612	(0.568, 0.656)	1.077	(0.893, 1.261)
192	62.171	(56.461, 67.882)	0.314	(0.285, 0.343)	0.524	(0.505, 0.543)
384	61.051	(51.638, 70.464)	0.155	(0.131, 0.178)	0.244	(0.208, 0.281)
Sierra						
12	46.988	(45.605, 48.371)	3.914	(3.799, 4.030)	4.173	(4.047, 4.299)
24	60.227	(57.508, 62.945)	2.507	(2.394, 2.621)	2.693	(2.578, 2.809)
48	71.567	(67.811, 75.323)	1.489	(1.411, 1.568)	1.590	(1.507, 1.674)
96	73.738	(67.218, 80.259)	0.767	(0.699, 0.834)	0.836	(0.757, 0.915)
192	108.503	(101.225, 115.780)	0.559	(0.521, 0.596)	0.620	(0.577, 0.663)
384	163.115	(155.688, 170.541)	0.413	(0.394, 0.433)	0.513	(0.469, 0.557)
768	177.109	(170.777, 183.441)	0.225	(0.219, 0.232)	0.273	(0.273, 0.274)
1536	159.163	(152.303, 166.023)	0.102	(0.098, 0.107)	0.121	(0.119, 0.122)
BG/P						
32	141.436	(135.834, 147.037)	4.802	(4.611, 4.992)	29.302	(28.141, 30.462)
64	255.139	(245.034, 265.244)	4.328	(4.157, 4.500)	26.960	(25.892, 28.028)
128	424.403	(407.594, 441.211)	3.583	(3.441, 3.725)	23.886	(22.940, 24.832)
256	502.093	(482.207, 521.978)	2.077	(1.995, 2.159)	12.171	(11.689, 12.653)
512	490.853	(471.412, 510.293)	1.016	(0.975, 1.056)	5.079	(4.877, 5.280)
1024	246.878	(237.101, 256.656)	0.250	(0.240, 0.260)	2.398	(2.303, 2.493)

Table B.1: Perceived MPI, effective MPI and effective POSIX bandwidths for IOR through HDF-5 on Minerva, Sierra and BG/P.

Numeric Data for Perceived and Effective Bandwidth

Cores	Perceived MPI		Effective MPI		Effective POSIX	
	B/W	95% CI	B/W	95% CI	B/W	95% CI
Minerva						
12	127.875	(119.601, 136.148)	10.655	(9.966, 11.345)	18.465	(16.925, 20.004)
24	109.681	(96.938, 122.424)	4.569	(4.039, 5.100)	9.175	(8.112, 10.239)
48	131.647	(119.650, 143.645)	2.742	(2.493, 2.992)	6.139	(5.394, 6.883)
96	142.439	(130.622, 154.256)	1.483	(1.360, 1.606)	3.201	(3.043, 3.360)
192	142.493	(131.716, 153.270)	0.742	(0.686, 0.798)	1.546	(1.433, 1.659)
384	164.584	(157.378, 171.790)	0.429	(0.410, 0.447)	0.874	(0.844, 0.903)
Sierra						
12	207.496	(159.452, 255.540)	17.290	(13.287, 21.293)	33.002	(25.311, 40.692)
24	220.744	(163.499, 277.989)	9.197	(6.812, 11.581)	19.090	(13.827, 24.354)
48	205.559	(164.239, 246.880)	4.282	(3.422, 5.143)	10.768	(8.950, 12.586)
96	227.694	(225.807, 229.580)	2.371	(2.349, 2.392)	7.015	(6.333, 7.698)
192	234.375	(209.612, 259.138)	1.221	(1.092, 1.350)	3.714	(3.545, 3.883)
384	274.170	(241.217, 307.124)	0.714	(0.628, 0.800)	2.532	(2.187, 2.876)
768	273.953	(256.134, 291.771)	0.357	(0.333, 0.380)	1.327	(1.251, 1.403)
1536	308.307	(285.644, 330.969)	0.201	(0.186, 0.215)	0.622	(0.592, 0.652)
BG/P						
32	141.099	(135.511, 46.687)	4.421	(4.246, 4.596)	30.101	(28.909, 31.293)
64	247.044	(237.260, 256.829)	3.870	(3.717, 4.023)	26.799	(25.738, 27.860)
128	402.305	(386.371, 418.238)	3.150	(3.025, 3.274)	23.680	(22.742, 24.617)
256	472.961	(454.229, 491.693)	1.851	(1.777, 1.924)	12.405	(11.914, 12.896)
512	483.548	(464.397, 502.699)	0.949	(0.911, 0.986)	5.659	(5.435, 5.883)
1024	245.878	(236.140, 255.616)	0.240	(0.231, 0.250)	2.298	(2.207, 2.389)

Table B.2: Perceived MPI, effective MPI and effective POSIX bandwidths for IOR through MPI-IO on Minerva, Sierra and BG/P.

Cores	Perceived MPI		Effective MPI		Effective POSIX	
	B/W	95% CI	B/W	95% CI	B/W	95% CI
Minerva						
12	47.618	(44.166, 51.070)	3.965	(3.678, 4.252)	12.387	(8.558, 16.217)
24	41.117	(37.282, 44.952)	1.693	(1.543, 1.844)	3.181	(3.041, 3.322)
48	58.350	(51.025, 65.674)	1.205	(1.054, 1.357)	1.840	(1.587, 2.093)
96	64.538	(50.901, 78.176)	0.663	(0.520, 0.806)	0.971	(0.767, 1.176)
192	80.422	(74.442, 86.401)	0.412	(0.382, 0.443)	0.578	(0.531, 0.625)
384	80.507	(75.378, 85.635)	0.206	(0.190, 0.222)	0.292	(0.257, 0.328)
Sierra						
12	147.348	(142.026, 152.669)	10.616	(10.080, 11.152)	27.228	(23.044, 31.412)
24	134.902	(119.046, 150.757)	5.511	(4.890, 6.132)	12.541	(10.025, 15.058)
48	154.352	(145.217, 163.488)	3.098	(2.903, 3.293)	5.483	(5.281, 5.685)
96	135.228	(128.708, 141.748)	1.385	(1.322, 1.447)	2.218	(2.151, 2.285)
192	132.892	(127.572, 138.212)	0.685	(0.658, 0.712)	1.179	(1.152, 1.206)
384	125.428	(121.639, 129.217)	0.323	(0.313, 0.333)	0.528	(0.507, 0.548)
768	134.317	(126.149, 142.485)	0.172	(0.161, 0.182)	0.246	(0.238, 0.255)
1536	129.882	(118.563, 141.200)	0.084	(0.079, 0.090)	0.106	(0.100, 0.112)
BG/P						
32	172.849	(166.003, 179.695)	5.635	(5.412, 5.858)	28.985	(27.837, 30.133)
64	278.380	(267.355, 289.405)	4.515	(4.336, 4.694)	23.845	(22.901, 24.790)
128	361.356	(347.044, 375.667)	2.903	(2.788, 3.018)	19.857	(19.070, 20.643)
256	555.880	(533.864, 577.896)	2.220	(2.132, 2.308)	15.836	(15.209, 16.463)
512	579.125	(556.188, 602.061)	1.151	(1.106, 1.197)	7.332	(7.041, 7.622)
1024	213.647	(205.185, 222.109)	0.210	(0.201, 0.218)	2.148	(2.063, 2.233)

Table B.3: Perceived MPI, effective MPI and effective POSIX bandwidths for FLASH-IO through HDF-5 on Minerva, Sierra and BG/P.

Numeric Data for Perceived and Effective Bandwidth

Cores	Perceived MPI		Effective MPI		Effective POSIX	
	B/W	95% CI	B/W	95% CI	B/W	95% CI
Minerva						
1	680.528	(645.412, 715.643)	680.528	(645.412, 715.643)	1134.095	(1041.052, 1227.139)
4	375.478	(333.395, 417.561)	94.264	(83.570, 104.958)	605.905	(499.718, 712.092)
16	252.887	(203.750, 302.025)	15.883	(12.795, 18.972)	218.024	(170.025, 266.023)
64	233.456	(214.638, 252.275)	3.651	(3.356, 3.946)	80.091	(71.018, 89.163)
256	173.696	(163.691, 183.700)	0.678	(0.639, 0.718)	19.049	(17.748, 20.349)
Sierra						
1	220.643	(210.264, 231.022)	220.643	(210.264, 231.022)	235.509	(222.126, 248.892)
4	210.142	(191.608, 228.675)	52.576	(47.922, 57.229)	294.799	(265.016, 324.582)
16	212.486	(164.552, 260.420)	13.346	(10.324, 16.368)	155.754	(123.069, 188.439)
64	126.102	(120.729, 131.474)	1.970	(1.887, 2.054)	41.970	(38.573, 45.368)
256	115.191	(107.091, 123.291)	0.450	(0.418, 0.482)	7.977	(7.511, 8.443)
1024	96.632	(88.439, 104.825)	0.094	(0.086, 0.102)	1.555	(1.443, 1.667)
BG/P						
16	84.149	(80.816, 87.482)	5.373	(5.161, 5.586)	132.102	(126.870, 137.334)
64	151.457	(145.458, 157.455)	2.877	(2.763, 2.991)	49.056	(47.113, 50.999)
256	278.103	(267.089, 289.118)	8.578	(8.238, 8.917)	21.102	(20.266, 21.938)
1024	504.126	(484.160, 524.092)	3.447	(3.310, 3.583)	8.746	(8.400, 9.093)

Table B.4: Perceived MPI, effective MPI and effective POSIX bandwidths for BT class C on Minerva, Sierra and BG/P.

Cores	Perceived MPI		Effective MPI		Effective POSIX	
	B/W	95% CI	B/W	95% CI	B/W	95% CI
Minerva						
16	304.569	(198.904, 410.234)	19.044	(12.338, 25.750)	444.626	(394.055, 495.197)
64	437.006	(431.559, 442.452)	6.830	(6.745, 6.916)	167.190	(153.586, 180.794)
256	382.567	(364.322, 400.813)	1.494	(1.423, 1.565)	47.438	(43.525, 51.351)
Sierra						
16	155.918	(146.865, 164.971)	9.760	(8.629, 10.892)	122.848	(116.058, 129.637)
64	135.191	(127.960, 142.422)	2.113	(2.000, 2.226)	59.092	(53.286, 64.898)
256	141.174	(136.546, 145.803)	0.551	(0.533, 0.570)	13.832	(13.603, 14.061)
1024	133.270	(130.813, 135.728)	0.130	(0.128, 0.133)	2.874	(2.698, 3.049)
4096	118.793	(109.880, 127.705)	0.029	(0.027, 0.031)	0.585	(0.543, 0.628)

Table B.5: Perceived MPI, effective MPI and effective POSIX bandwidths for BT class D on Minerva and Sierra.

APPENDIX C

FLASH-IO Analysis and Optimisation Data

	12	24	48	96	192	384
Data Written (MB)	2460.540	4921.067	9842.121	19684.229	39368.445	78736.878
MPI_File_write() calls	373	733	1453	2893	5773	11533
POSIX write() calls	5173	10333	20653	41293	82573	165133
POSIX read() calls	5088	10176	20352	40704	81408	162816
Locks requested	5088	10176	20352	40704	81408	162816
MPI write time (s)	623.911	2929.945	8320.767	31598.843	95974.556	384706.897
POSIX write time (s)	218.345	1550.154	5467.534	21533.644	68600.040	274331.768
POSIX read time (s)	220.656	885.581	2474.529	9005.330	26156.646	108415.408
Lock time (s)	183.823	485.925	374.036	1050.601	1199.257	1922.617
Unlock time (s)	0.100	6.292	0.861	1.698	3.439	6.732

Table C.1: MPI and POSIX function statistics for FLASH-IO on Minerva.

	12	24	48	96
Data Written (MB)	2460.540	4921.067	9842.121	19684.229
MPI_File_write() calls	373	743	1453	2893
POSIX write() calls	5173	10333	20653	41293
POSIX read() calls	5088	10176	20352	40704
Locks requested	5088	10176	20352	40704
MPI write time (s)	232.368	905.980	3190.456	14248.187
POSIX write time (s)	2460.540	4921.067	9842.121	19684.229
POSIX read time (s)	2118.813	4491.588	9108.378	18388.578
Lock time (s)	2.492	5.993	12.581	26.313
Unlock time (s)	2.486	5.953	11.853	21.604

Table C.2: MPI and POSIX function statistics for FLASH-IO on Sierra 12 to 96 cores.

	192	384	768	1536
Data Written (MB)	39368.445	78736.878	157473.742	314947.472
MPI_File_write() calls	7633	15203	27112	46942
POSIX write() calls	82573	165133	330252	660492
POSIX read() calls	81408	162816	325632	651264
Locks requested	81408	162816	325632	651264
MPI write time (s)	57538.796	244006.624	918510.086	3777346.881
POSIX write time (s)	39368.445	78736.878	157473.742	314947.472
POSIX read time (s)	38071.639	75773.789	155339.023	313195.438
Lock time (s)	74.785	198.840	615.579	2409.944
Unlock time (s)	64.528	121.028	304.644	1057.103

Table C.3: MPI and POSIX function statistics for FLASH-IO on Sierra 192 to 1536 cores.

	32	64	128	256	512	1024
Data Written (MB)	6558.887	13120.292	26243.103	52488.725	104979.968	209962.454
MPI_File_write() calls	1011	1971	3891	7731	15411	30771
POSIX write() calls	1971	3891	7731	15411	30771	61491
POSIX read() calls	0	0	0	0	0	0
Locks requested	1971	3891	7731	15411	30771	61491
MPI write time (s)	1163.928	2905.728	9039.741	23642.281	91188.504	1001093.819
POSIX write time (s)	226.287	550.226	1321.619	3314.475	14318.643	97742.674
POSIX read time (s)	0.000	0.000	0.000	0.000	0.000	0.000
Lock time (s)	24.593	47.661	103.852	136.110	251.895	1012.799
Unlock time (s)	3.877	8.003	17.224	22.775	27.985	49.410

Table C.4: MPI and POSIX function statistics for FLASH-IO on BG/P.

Cores	B/W	Original 95% CI	B/W	DS off 95% CI	B/W	DS off, CB on 95% CI
Minerva						
12	119.725	(115.730, 123.720)	259.342	(251.667, 267.017)	231.090	(187.192, 274.989)
24	126.640	(110.132, 143.148)	248.891	(217.405, 280.377)	262.451	(252.218, 272.684)
48	129.594	(120.512, 138.676)	260.645	(244.414, 276.876)	283.877	(271.360, 296.393)
96	141.710	(138.635, 144.785)	266.396	(241.769, 291.023)	302.803	(289.431, 316.174)
192	141.796	(130.662, 152.930)	265.169	(254.324, 276.014)	309.424	(287.468, 331.380)
384	137.442	(133.356, 141.528)	268.473	(256.129, 280.816)	310.508	(302.516, 318.501)
Sierra						
12	268.676	(238.434, 298.918)	352.054	(330.295, 373.813)	334.271	(320.138, 348.404)
24	199.376	(175.188, 223.564)	276.024	(258.802, 293.246)	344.338	(316.827, 371.849)
48	222.221	(220.018, 224.423)	318.886	(306.690, 331.081)	404.295	(385.334, 423.256)
96	240.606	(217.310, 263.902)	362.012	(336.092, 387.932)	471.701	(448.554, 494.848)
192	298.036	(291.031, 305.040)	467.650	(436.188, 499.112)	549.125	(523.263, 574.986)
384	275.910	(261.112, 290.708)	501.472	(463.454, 539.490)	448.509	(427.101, 469.917)

Table C.5: FLASH-IO performance on Minerva and Sierra with collective buffering and data sieving optimisation options.

APPENDIX D

LDPLFS Source Code Examples

```
ssize_t write(int fd, const void *buf, size_t count) {
    ssize_t ret;

    // check if fd is a plfs file or a normal file
    if (plfs_files.find(fd) != plfs_files.end()) {
        // if the file is a plfs file,
        // find its current virtual offset
        off_t offset = lseek(fd, 0, SEEK_CUR);
        // perform plfs write operation
        ret = plfs_write(plfs_files.find(fd)->second->fd,
            (const char *) buf, count, offset, getpid());

        // update the virtual offset
        lseek(fd, ret, SEEK_CUR);
    } else {
        // perform a standard write on a normal file
        ret = __real_write(fd, buf, count);
    }
    return ret;
}
```

Listing D.1: Source code for the `write()` function in LDPLFS.

```
int close(int fd) {
    // check if fd is a plfs file or not
    if (plfs_files.find(fd) != plfs_files.end()) {
        // only close the PLFS file if fd hasn't been duplicated
        if (!isDuplicated(fd)) {
            // close the file and remove it from the book keeping
            plfs_close(plfs_files.find(fd)->second->fd, getpid(),
                getuid(), plfs_files.find(fd)->second->mode,
                NULL);
            delete plfs_files.find(fd)->second->path;
            delete plfs_files.find(fd)->second;
        }
        // remove fd from book keeping
        plfs_files.erase(fd);
    }
    // close either a real file or the 'virtual file'
    int ret = __real_close(fd);
    return ret;
}
```

Listing D.2: Source code the `close()` function in LDPLFS.

```

int open(const char *path, int flags, ...) {
    int ret;
    char *cpath = resolvePath(path);

    // determine if the path given is in a plfs mount
    if (is_plfs_path(cpath)) {
        mode_t mode;

        if ((flags & O_CREAT) == O_CREAT) {
            va_list argf;
            va_start(argf, flags);
            mode = va_arg(argf, mode_t);
            va_end(argf);
        } else {
            int m = plfs_mode(cpath, &mode);
        }

        // create a plfs file pointer
        plfs_file *tmp = new plfs_file();

        // open the given file using the plfs open command
        int err = plfs_open(&(tmp->fd), cpath,
                           flags, getpid(), mode, NULL);
        // in the event of an error, set errno correctly.
        if (err != 0) {
            errno = -err; // invert errorcode correctly.
            ret = -1;
            delete tmp;
        } else {
            // create a tmp file to store seek information
            ret = fileno(__real_tmpfile());

            tmp->path = new std::string(cpath);
            tmp->mode = flags;

            // add the pairing to a hash table
            plfs_files.insert(
                std::pair<int, plfs_file *>(ret, tmp));
        }
    } else {
        // treat as a standard file
        if ((flags & O_CREAT) == O_CREAT) {
            va_list argf;
            va_start(argf, flags);
            mode_t mode = va_arg(argf, mode_t);
            va_end(argf);
            ret = __real_open(path, flags, mode);
        } else {
            ret = __real_open(path, flags);
        }
    }

    free(cpath);

    return ret;
}

```

Listing D.3: Source code for the open() function in LDPLFS.

APPENDIX E

LDPLFS Numeric Data

Nodes	B/W	ad_ufs 95% CI		B/W	PLFS 95% CI
Read					
1	174.153	(153.315, 194.991)	FUSE	132.702	(122.900, 142.503)
			ad_plfs	184.471	(169.849, 199.094)
			LDPLFS	170.047	(149.238, 190.856)
2	180.987	(176.741, 185.234)	FUSE	173.380	(160.506, 186.254)
			ad_plfs	194.687	(179.015, 210.360)
			LDPLFS	193.223	(172.575, 213.871)
4	200.636	(194.517, 206.756)	FUSE	205.273	(195.698, 214.848)
			ad_plfs	216.168	(209.079, 223.257)
			LDPLFS	209.455	(202.026, 216.884)
8	204.591	(199.800, 209.381)	FUSE	216.165	(212.248, 220.083)
			ad_plfs	216.047	(212.250, 219.845)
			LDPLFS	218.693	(215.405, 221.981)
16	197.569	(179.152, 215.987)	FUSE	200.321	(176.539, 224.103)
			ad_plfs	204.896	(182.338, 227.453)
			LDPLFS	217.068	(215.258, 218.878)
32	200.377	(188.224, 212.531)	FUSE	193.445	(176.928, 209.962)
			ad_plfs	195.587	(179.690, 211.485)
			LDPLFS	208.442	(192.302, 224.583)
64	192.005	(185.070, 198.940)	FUSE	202.738	(198.525, 206.951)
			ad_plfs	201.505	(192.903, 210.107)
			LDPLFS	215.595	(213.859, 217.331)
Write					
1	115.951	(104.492, 127.410)	FUSE	62.947	(58.815, 67.079)
			ad_plfs	138.206	(117.090, 159.322)
			LDPLFS	106.517	(70.634, 142.399)
2	131.026	(103.939, 158.114)	FUSE	91.774	(64.647, 118.901)
			ad_plfs	148.143	(138.184, 158.103)
			LDPLFS	155.341	(136.122, 174.559)
4	123.323	(96.784, 149.862)	FUSE	108.932	(103.186, 114.677)
			ad_plfs	131.954	(126.243, 137.666)
			LDPLFS	136.569	(131.707, 141.431)
8	153.378	(127.400, 179.357)	FUSE	124.776	(98.018, 151.533)
			ad_plfs	176.427	(169.889, 182.965)
			LDPLFS	153.583	(125.570, 181.596)
16	167.082	(162.040, 172.124)	FUSE	139.542	(122.410, 156.675)
			ad_plfs	184.252	(178.311, 190.192)
			LDPLFS	159.486	(138.883, 180.088)
32	145.436	(136.966, 153.907)	FUSE	137.018	(127.210, 146.826)
			ad_plfs	183.427	(178.636, 188.219)
			LDPLFS	173.023	(161.362, 184.684)
64	154.127	(141.509, 166.745)	FUSE	122.871	(120.504, 125.237)
			ad_plfs	186.549	(181.085, 192.013)
			LDPLFS	190.150	(186.756, 193.544)

Table E.1: Read and write performance of PLFS through FUSE, the `ad_plfs` MPI-IO driver and LDPLFS compared to the standard `ad_ufs` MPI-IO driver on Minerva, using 1 core per node.

Nodes	B/W	ad.ufs		B/W	PLFS	
		95% CI			95% CI	
Read						
1	76.458	(69.306, 83.609)	FUSE	51.996	(50.600, 53.393)	
			ad_plfs	169.281	(161.425, 177.137)	
			LDPLFS	178.569	(172.671, 184.466)	
2	112.283	(105.157, 119.410)	FUSE	79.251	(76.315, 82.188)	
			ad_plfs	205.034	(197.616, 212.453)	
			LDPLFS	207.050	(192.733, 221.367)	
4	140.725	(136.685, 144.765)	FUSE	118.506	(112.562, 124.450)	
			ad_plfs	216.213	(212.995, 219.431)	
			LDPLFS	213.887	(208.873, 218.900)	
8	151.341	(144.449, 158.233)	FUSE	156.072	(151.289, 160.855)	
			ad_plfs	215.321	(211.779, 218.864)	
			LDPLFS	219.013	(217.735, 220.290)	
16	164.695	(156.390, 172.999)	FUSE	182.058	(180.030, 184.087)	
			ad_plfs	213.500	(205.120, 221.881)	
			LDPLFS	214.439	(211.739, 217.139)	
32	160.321	(145.501, 175.140)	FUSE	191.395	(182.814, 199.976)	
			ad_plfs	218.613	(217.969, 219.256)	
			LDPLFS	218.425	(216.898, 219.953)	
64	173.786	(163.189, 184.383)	FUSE	193.368	(182.256, 204.480)	
			ad_plfs	219.345	(216.882, 221.809)	
			LDPLFS	217.369	(215.761, 218.977)	
Write						
1	58.826	(43.887, 73.766)	FUSE	59.829	(56.688, 62.971)	
			ad_plfs	131.404	(117.308, 145.500)	
			LDPLFS	100.627	(91.149, 110.104)	
2	51.981	(42.369, 61.594)	FUSE	69.101	(65.868, 72.335)	
			ad_plfs	91.449	(77.016, 105.882)	
			LDPLFS	92.688	(89.200, 96.177)	
4	75.159	(68.094, 82.225)	FUSE	86.946	(84.810, 89.081)	
			ad_plfs	109.437	(100.339, 118.535)	
			LDPLFS	103.375	(94.687, 112.063)	
8	82.866	(76.931, 88.801)	FUSE	105.507	(102.064, 108.949)	
			ad_plfs	133.665	(125.743, 141.587)	
			LDPLFS	125.284	(122.299, 128.269)	
16	131.388	(123.676, 139.099)	FUSE	113.819	(112.163, 115.475)	
			ad_plfs	163.389	(157.067, 169.711)	
			LDPLFS	140.184	(138.104, 142.263)	
32	128.644	(104.467, 152.821)	FUSE	113.880	(107.410, 120.351)	
			ad_plfs	174.800	(161.922, 187.679)	
			LDPLFS	158.293	(156.781, 159.805)	
64	160.263	(151.144, 169.382)	FUSE	90.344	(88.655, 92.033)	
			ad_plfs	184.219	(171.440, 196.998)	
			LDPLFS	163.318	(161.144, 165.492)	

Table E.2: Read and write performance of PLFS through FUSE, the `ad_plfs` MPI-IO driver and LDPLFS compared to the standard `ad_ufs` MPI-IO driver on Minerva, using 2 cores per node.

Nodes	B/W	ad_ufs		B/W	PLFS	
		95% CI			95% CI	
Read						
1	87.556	(74.016, 101.097)	FUSE	72.647	(68.509, 76.785)	
			ad_plfs	159.387	(150.973, 167.800)	
			LDPLFS	132.534	(98.890, 166.178)	
2	110.367	(99.314, 121.420)	FUSE	113.900	(110.046, 117.754)	
			ad_plfs	185.589	(163.692, 207.486)	
			LDPLFS	174.380	(146.030, 202.730)	
4	131.679	(115.139, 148.218)	FUSE	151.274	(141.944, 160.604)	
			ad_plfs	209.524	(202.633, 216.416)	
			LDPLFS	186.016	(166.381, 205.652)	
8	149.701	(139.861, 159.542)	FUSE	115.713	(112.006, 119.420)	
			ad_plfs	212.314	(206.274, 218.354)	
			LDPLFS	189.479	(170.760, 208.198)	
16	163.049	(156.890, 169.208)	FUSE	175.760	(171.247, 180.272)	
			ad_plfs	210.180	(204.446, 215.914)	
			LDPLFS	202.981	(192.125, 213.837)	
32	171.889	(163.390, 180.387)	FUSE	165.813	(153.764, 177.862)	
			ad_plfs	209.694	(204.347, 215.040)	
			LDPLFS	198.946	(186.768, 211.124)	
64	165.700	(147.207, 184.194)	FUSE	167.750	(159.057, 176.444)	
			ad_plfs	215.947	(212.615, 219.280)	
			LDPLFS	206.272	(199.712, 212.833)	
Write						
1	88.878	(79.106, 98.650)	FUSE	58.888	(50.195, 67.580)	
			ad_plfs	118.508	(98.583, 138.433)	
			LDPLFS	89.247	(66.860, 111.635)	
2	101.847	(84.651, 119.043)	FUSE	62.316	(60.470, 64.162)	
			ad_plfs	132.943	(100.633, 165.253)	
			LDPLFS	99.963	(91.592, 108.333)	
4	84.651	(77.047, 92.254)	FUSE	60.640	(47.694, 73.586)	
			ad_plfs	129.401	(116.887, 141.915)	
			LDPLFS	108.328	(98.829, 117.827)	
8	82.580	(69.041, 96.118)	FUSE	52.533	(48.067, 56.999)	
			ad_plfs	146.761	(135.788, 157.733)	
			LDPLFS	131.950	(117.438, 146.461)	
16	111.891	(104.910, 118.872)	FUSE	76.339	(71.192, 81.485)	
			ad_plfs	162.167	(153.775, 170.558)	
			LDPLFS	146.053	(134.681, 157.425)	
32	122.804	(119.148, 126.461)	FUSE	125.411	(122.552, 128.270)	
			ad_plfs	159.126	(154.316, 163.937)	
			LDPLFS	147.484	(142.124, 152.844)	
64	131.420	(125.213, 137.628)	FUSE	104.973	(96.186, 113.760)	
			ad_plfs	176.809	(169.651, 183.966)	
			LDPLFS	158.409	(155.963, 160.855)	

Table E.3: Read and write performance of PLFS through FUSE, the `ad_plfs` MPI-IO driver and LDPLFS compared to the standard `ad_ufs` MPI-IO driver on Minerva, using 4 cores per node.

Cores	B/W	ad_ufs			B/W	PLFS	
		95% CI				95% CI	
4	317.570	(265.909, 369.231)	ad_plfs	330.930	339.713	(330.028, 331.832)	
			LDPLFS			(313.237, 366.190)	
16	449.537	(438.342, 460.731)	ad_plfs	613.103	628.990	(565.199, 661.008)	
			LDPLFS			(586.299, 671.681)	
64	390.245	(353.780, 426.710)	ad_plfs	1653.827	1487.917	(1546.650, 1761.004)	
			LDPLFS			(1426.418, 1549.415)	
256	327.010	(326.834, 327.186)	ad_plfs	3722.670	2846.797	(3527.516, 3917.824)	
			LDPLFS			(2409.288, 3284.305)	
1024	264.995	(255.185, 274.805)	ad_plfs	3021.910	3074.595	(1929.717, 4114.103)	
			LDPLFS			(2388.605, 3760.585)	

Table E.4: Write performance in BT class C for PLFS through the `ad_plfs` MPI-IO driver and LDPLFS compared to the standard `ad_ufs` MPI-IO driver on Sierra.

Cores	B/W	ad_ufs			B/W	PLFS	
		95% CI				95% CI	
64	321.480	(180.948, 462.012)	ad_plfs	1348.010	1155.093	(1280.434, 1415.586)	
			LDPLFS			(996.537, 1313.650)	
256	449.895	(390.521, 509.269)	ad_plfs	2876.855	2211.220	(2711.829, 3041.881)	
			LDPLFS			(2085.182, 2337.258)	
1024	251.987	(239.848, 264.125)	ad_plfs	264.223	231.193	(228.386, 300.060)	
			LDPLFS			(160.736, 301.651)	
4096	284.315	(273.173, 295.457)	ad_plfs	1565.760	1548.845	(1554.471, 1577.049)	
			LDPLFS			(1526.982, 1570.708)	

Table E.5: Write performance in BT class D for PLFS through the `ad_plfs` MPI-IO driver and LDPLFS compared to the standard `ad_ufs` MPI-IO driver on Sierra.

Nodes	B/W	ad_ufs			B/W	PLFS	
		95% CI				95% CI	
1	332.150	(323.457, 340.842)	ad_plfs	282.513		(272.341, 292.685)	
			LDPLFS	308.306		(293.990, 322.622)	
2	234.704	(207.942, 261.466)	ad_plfs	677.992		(658.070, 697.913)	
			LDPLFS	663.059		(540.204, 585.914)	
4	323.962	(280.407, 367.516)	ad_plfs	1073.471	(1032.116, 1114.825)		
			LDPLFS	1023.501	(965.397, 1081.605)		
8	382.921	(368.875, 396.967)	ad_plfs	1468.949	(1405.361, 1532.538)		
			LDPLFS	1458.562	(1375.307, 1541.817)		
16	406.641	(374.487, 438.794)	ad_plfs	1642.854	(1548.991, 1736.718)		
			LDPLFS	1632.436	(1575.404, 1689.467)		
32	439.282	(422.463, 456.102)	ad_plfs	1284.342	(1238.790, 1329.894)		
			LDPLFS	1224.472	(1192.328, 1256.616)		
64	464.171	(427.776, 500.566)	ad_plfs	709.821	(698.345, 721.296)		
			LDPLFS	714.732	(707.204, 722.260)		
128	477.430	(465.562, 489.298)	ad_plfs	201.701	(95.634, 307.769)		
			LDPLFS	378.024	(365.059, 390.988)		
256	518.736	(485.697, 551.776)	ad_plfs	183.701	(130.657, 236.746)		
			LDPLFS	204.371	(195.527, 213.215)		

Table E.6: Write performance in FLASH-IO for PLFS through the `ad_plfs` MPI-IO driver and LDPLFS compared to the standard `ad_ufs` MPI-IO driver on Sierra.

APPENDIX F

Optimality Search Numeric Data

Stripe Size (MB)	1	2	4	8	Stripe Count				
					16	32	64	128	160
1	242.20	313.02	562.23	972.19	1202.30	1932.38	2910.96	4071.30	4074.45
2	216.35	318.84	544.99	936.85	1365.31	2343.35	3899.43	5310.18	6009.99
4	212.47	278.69	591.76	1016.44	1620.91	2916.30	4280.74	5852.94	7956.12
8	222.50	307.35	607.45	1011.24	1786.70	3318.84	4696.76	7106.03	9467.26
16	215.28	347.28	539.68	1135.96	2155.92	3899.88	6152.11	8684.13	11285.07
32	192.90	284.25	654.65	1339.34	2407.39	4051.20	6745.11	11288.24	13884.42
64	216.00	398.92	764.62	1522.98	3057.95	4079.76	7770.03	13621.08	14768.08
128	229.82	387.00	767.69	1492.31	2993.44	4063.87	7114.70	13799.54	15609.38
256	192.05	394.92	808.21	1554.25	3006.16	4250.09	8005.93	13519.03	14753.50

Table F.1: Numerical data for Figure 6.2, displaying bandwidth achieved by IOR on 1,024 cores, while varying Lustre stripe size and stripe count.

Tasks	B/W	Achieved		B/W	Ideal	
			95% CI			95% CI
1	211.265	(156.310, 266.220)		211.265	(156.310, 266.220)	
2	90.777	(76.506, 105.049)		105.632	(78.155, 133.110)	
3	84.758	(64.103, 105.413)		70.422	(52.103, 88.740)	
4	54.092	(43.168, 65.017)		52.816	(39.077, 66.555)	
5	42.952	(35.381, 50.523)		42.253	(31.262, 53.244)	
6	36.591	(30.655, 42.528)		35.211	(26.052, 44.370)	
7	29.626	(21.640, 37.611)		30.181	(22.330, 38.031)	
8	25.393	(24.602, 26.184)		26.408	(19.539, 33.277)	
9	21.458	(18.787, 24.129)		23.474	(17.368, 29.580)	
10	17.964	(16.566, 19.362)		21.126	(15.631, 26.622)	
11	16.608	(15.237, 17.978)		19.206	(14.210, 24.202)	
12	15.939	(13.847, 18.030)		17.605	(13.026, 22.185)	
13	14.231	(12.610, 15.852)		16.251	(12.024, 20.478)	
14	13.299	(12.245, 14.352)		15.090	(11.165, 19.016)	
15	12.768	(11.977, 13.559)		14.084	(10.421, 17.748)	
16	10.048	(9.446, 10.650)		13.204	(9.769, 16.639)	

Table F.2: Numerical data for Figure 6.3, displaying bandwidth per task under contention, along with the idealised values.

APPENDIX G

PLFS Performance and Stripe Collision Data

Collisions	Experiment				
	1	2	3	4	5
0	32	32	32	32	32
D_{inuse}	32	32	32	32	32
D_{load}	1.00	1.00	1.00	1.00	1.00
BW (MB/s)	424.67	802.4	1175.22	259.56	1102.96

Table G.1: Stripe collision statistics for PLFS backend directory running with 16 cores.

Collisions	Experiment				
	1	2	3	4	5
0	60	52	57	64	57
1	2	6	2	0	2
2	0	0	1	0	1
D_{inuse}	62	58	60	64	60
D_{load}	1.03	1.10	1.07	1.0	1.07
BW (MB/s)	1052.73	562.23	640.72	736.52	644.43

Table G.2: Stripe collision statistics for PLFS backend directory running with 32 cores.

Collisions	Experiment				
	1	2	3	4	5
0	85	85	101	94	95
1	20	17	12	14	15
2	1	3	1	2	1
D_{inuse}	106	105	114	110	111
D_{load}	1.21	1.22	1.12	1.16	1.15
BW (MB/s)	1158.81	2067.78	948.45	3903.84	804.60

Table G.3: Stripe collision statistics for PLFS backend directory running with 64 cores.

PLFS Performance and Stripe Collision Data

Collisions	Experiment				
	1	2	3	4	5
0	155	162	166	153	147
1	40	36	36	41	47
2	7	6	6	7	5
3	0	1	0	0	0
D_{inuse}	202	205	208	201	199
D_{load}	1.27	1.25	1.23	1.27	1.29
BW (MB/s)	6799.10	1555.19	1894.95	1904.80	6919.04

Table G.4: Stripe collision statistics for PLFS backend directory running with 128 cores.

Collisions	Experiment				
	1	2	3	4	5
0	192	192	180	175	183
1	110	101	92	99	102
2	18	29	35	37	32
3	9	5	7	7	6
4	2	1	3	0	1
5	0	1	0	0	0
D_{inuse}	331	329	317	318	324
D_{load}	1.55	1.56	1.62	1.61	1.58
BW (MB/s)	5403.23	9726.24	5635.64	11539.40	3329.90

Table G.5: Stripe collision statistics for PLFS backend directory running with 256 cores.

Collisions	Experiment				
	1	2	3	4	5
0	121	135	122	116	129
1	134	126	134	129	133
2	97	88	85	94	82
3	49	55	56	45	54
4	21	22	21	20	28
5	6	6	6	12	2
6	1	1	2	1	1
7	0	0	0	0	1
8	0	0	0	1	0
D_{inuse}	429	433	426	418	430
D_{load}	2.39	2.36	2.40	2.45	2.38
BW (MB/s)	12062.68	10469.38	10234.97	9768.07	11081.99

Table G.6: Stripe collision statistics for PLFS backend directory running with 512 cores.

PLFS Performance and Stripe Collision Data

Collisions	Experiment				
	1	2	3	4	5
0	32	33	35	36	24
1	53	61	64	59	62
2	97	82	75	80	95
3	90	92	98	84	103
4	82	77	80	76	71
5	53	64	62	59	62
6	34	34	27	40	26
7	20	12	16	27	19
8	9	8	12	3	9
9	3	6	6	5	5
10	2	4	1	1	1
D_{inuse}	475	473	476	470	477
D_{load}	4.31	4.33	4.30	4.36	4.29
BW (MB/s)	8524.22	8610.75	8754.29	8536.84	8449.57

Table G.7: Stripe collision statistics for PLFS backend directory running with 1,024 cores.

Collisions	Experiment				
	1	2	3	4	5
0	1	2	0	1	1
1	8	6	4	3	2
2	13	5	6	8	7
3	21	18	18	17	24
4	35	39	40	33	42
5	52	58	50	46	57
6	66	66	62	55	66
7	46	54	66	86	64
8	57	65	71	79	56
9	55	48	52	47	41
10	46	38	36	54	34
11	27	30	25	18	28
12	27	22	28	9	22
13	11	15	14	7	22
14	8	6	7	8	7
15	2	7	1	3	0
16	2	1	0	1	2
17	2	0	0	3	4
18	1	0	0	2	1
D_{inuse}	480	480	480	480	480
D_{load}	8.53	8.53	8.53	8.53	8.53
BW (MB/s)	5794.14	5585.98	5800.88	5592.34	5708.71

Table G.8: Stripe collision statistics for PLFS backend directory running with 2,048 cores.

Cores	ad.lustre		ad.plfs	
	B/W	95% CI	B/W	95% CI
16	403.748	(390.728, 416.768)	752.962	(398.413, 1107.511)
32	404.714	(393.092, 416.336)	727.326	(558.951, 895.701)
64	857.348	(832.819, 881.877)	1776.696	(648.889, 2904.503)
128	1987.512	(1908.244, 2066.780)	3814.616	(1375.185, 6254.047)
256	4354.983	(4288.692, 4421.273)	7126.882	(4159.661, 10094.103)
512	8985.136	(8777.611, 9192.661)	10723.418	(9947.064, 11499.772)
1024	13859.578	(12582.684, 15136.472)	8575.134	(8474.058, 8676.210)
2048	16200.156	(15441.574, 16958.738)	5696.410	(5604.855, 5787.965)
4096	16917.112	(16291.584, 17542.640)	3069.054	(3052.824, 3085.284)

Table G.9: Numeric data for Figure 6.6, showing the performance of IOR through Lustre and PLFS.